COMP50006 Compilers

Exercise 6: Common sub-expression elimination using available expressions

Introduction We naturally expect our compiler to spot where an expression is being recomputed unnecessarily.

If we look at expression trees, it's easy to imagine how we might identify repeated subtrees. However things look a bit more complicated when we look across sequences of assignments, in a program with complex control flow. In this exercise we explore how to identify common sub-expressions in three-address code.

We introduce "available expressions analysis". We aim to discover which expressions will have been computed by the time control arrives at each instruction in the program. We can then see whether we are recomputing an expression which is already available - that is, one which has already been evaluated, and whose value would be the same.

An expression is available at an instruction if its value has definitely been computed — and it has not been subsequently invalidated. It would be invalidated if an assignment to any of its variables might have been executed.

Note the care needed with "might" versus "must". The value *must* have been evaluated (so if control flow joins, we need the expression to be available on all incoming paths). It's not available if it *might* be invalidated along any path.

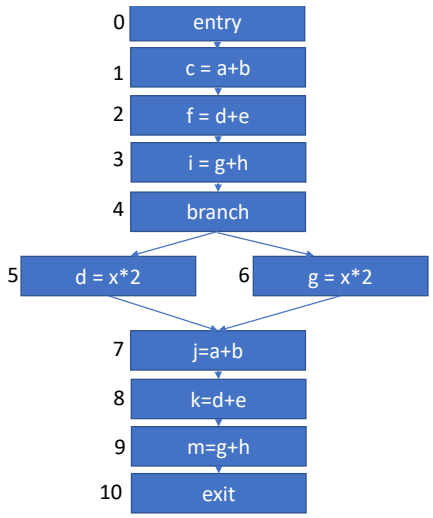
Three-address code For this exercise, we assume a compiler architecture with a "three-address code" intermediate representation (IR). This is a low-level control-flow-graph in which all operations have been broken down into the most primitive operations. The most complex operation is three-address arithmetic, of the form "$c = a \bigoplus b$". The name reflects that such an operation names at most three operands. This is a representation that comes *before* instruction selection, which may produce code that packs more than one primitive operation into each instruction.

Local common sub-expressions: straight-line code Here we see the "Avail" sets at the points before and after execution of a sequence of assignments in a three-address IR:

Three-address operation	Available expressions
	$\emptyset$
a = b+c	{b+c}
d = c+f	{b+c, c+f}
b = a-f	{c+f, a-f}
f = z*2	{z*2}

Initially the set is empty. After the assignment to a, we have an available expression "b+c". The assignment to d gives us another. The reassignment to b invalidates "b+c", but adds "a-f". The assignment to f invalidates both.

Global common sub-expressions: branches Consider this example:



Here it's clear that only "a+b" and "x*2" are available after the join - so only "j=a+b" can be eliminated.[1]

Straight-line code is clearly easier to analyse than general control flow. Some compilers (and textbooks) recommend breaking the code up into "basic blocks" — sequences of code with no branches and no labels. Then the CFG has basic blocks as nodes, and data flow analysis operates on basic blocks, rather than on primitive three-address operations. Doing this may help with compilation time - but complicates the presentation. Optimising basic blocks is sometimes

[1]The expression "x*2" represents a different opportunity - we can use a related dataflow analysis called "very busy expressions" for this.

described as "local" optimisation, while optimisation at the level of the CFG for a whole function is called "global".

Available expressions as a dataflow analysis problem We define, for each node n of the DFG, $AvailIn(n)$ and $AvailOut(n)$. These are sets of expressions - which for our purposes are never more complicated than "$a \bigoplus b$".

We define U to be the set of *all* the expressions in the program - in the case of the example above,

$$U = \{a + b, d + e, g + h, x * 2, \}$$

We know that the following must hold:

$$\begin{aligned} AvailIn(n) &= \bigcap_{p \in preds(n)} AvailOut(p) \\ AvailOut(n) &= gen(n) \cup (AvailIn(n) - kill(n)) \end{aligned}$$

Where $pred(n)$ is the set of predecessors of node n in the CFG. The crucial part is $gen(n)$ and $kill(n)$:

$gen(n)$ = the expression, if any, in the RHS of the instruction, assuming it's a simple arithmetic operation

and

$kill(n)$ = the set of expressions killed by n: all expressions in U that depend on the variable n assigns to.

We also know that

$$AvailOut(0) = \emptyset$$

Where node 0 is the entry point of the program's CFG.

The exercise

The challenge for you in this exercise is to write down the pseudocode to set up and solve for $AvailIn$ and $AvailOut$ for all the nodes in a program. You need to think about how to initialise the $AvailIn$ and $AvailOut$ sets, how to iterate, and how to detect termination.

(The level of detail intended here is as shown in the lecture slides, Chapter 6 part 2 slide 23 "Solving the dataflow equations").

Efficiency

(1) Should we visit the CFG nodes starting from the top, or from the bottom, as we did with live variable analysis? Why?

(2) Can you see how to implement your algorithm using bit-vectors - that is, using the bits in a 64-bit word to represent the $AvailIn$ and $AvailOut$ sets?

Using it

Write down the steps needed to use available expressions to actually optimise three-address code (in abstract terms).

Strategy

What performance consideration might lead you to *not* eliminate a common subexpression?

Paul Kelly *Imperial College* *January 2024*

Exercise 5: Common sub-expression elimination using available expressions

Sample solutions

The basic iteration structure is exactly what you expect - and very similar to what we did with live variable analysis. The subtlety is how we initialise the $AvailIn$ and $AvailOut$ sets. Suppose N is the set of all node ids:

$AvailOut(0) = \emptyset$
for $n \in N - \{0\}$
    $AvailIn(n) = AvailOut(n) = U$
do {
    for $n \in N - \{0\}$
        $AvailIn(n) = \bigcap_{p \in preds(n)} AvailOut(p)$
        $AvailOut(n) = gen(n) \cup (AvailIn(n) - kill(n))$
} while any $AvailOut(n)$ changes

As we iterate, information propagates through the control-flow graph, the $AvailIn$ and $AvailOut$ sets get *smaller*, until we have removed all the expressions that are not available at each point. We aim, at the end, to have the *largest* sets that satisfy the dataflow equations.

Efficiency

Available expressions is a *forward* analysis - information propagates in a forward direction. Thus, it makes sense to try to visit the nodes in a forward direction, in order to propagate information as fast as we can.

When we compute U, we can assign a bit index for each expression - and precompute the bitwise representation of each node's $kill(n)$.

Using it

You need to look at each node, check it's RHS, and see whether that expression is in that node's $AvailIn$ set. If so, find the instruction that generates it, and insert an instruction to copy the result to a new temporary register. You can then replace this node's RHS with that register.

Strategy

Common sub-expression requires the allocation of an additional register. If this were to cause spilling, it's unlikely that the optimisation would be profitable.

Beyond the course

Functions Suppose instead of primitive arithmetic operations "a*b" where a and b are scalars, we have expressions like "dotproduct(v,w)", where v and w are vectors. We would like to do similar optimisations - common sub-expression elimination, loop-invariant code motion etc. What does the compiler need to know about functions like "dotproduct" for this to work? A Haskell compiler can do this — what about in your (other) favourite languages?

Algebraic equivalences Consider "$d = a + b; e = d + c; f = b + c; g = a + f$". After execution, we should have that $e = (a + b) + c$, and $g = a + (b + c)$. Extending our algorithm to catch this is not so easy!

A more ambitious version of this is to go beyond associativity and commutativity, and to consider distributivity: consider

$$a * (b + c) \tag{1}$$

and

$$a * b + a * c \tag{2}$$

The former does less work. That latter exposes sub-expressions that might appear elsewhere in the program - or might be loop-invariant. When should we rewrite expressions like (1) to (2), and when should we rewrite the other way?

Associativity Find values for floating-point (`float`) variables a, b and c so that $a + (b + c) \neq (a + b) + c$.

Bonus: Find values for floating-point (`float`) variables a, b and c so that $\min(a, \min(b, c)) \neq \min(\min(a, b), c)$.

Paul Kelly *Imperial College* *January 2024*