Lecture Notes on Basic Optimizations

15-411: Compiler Design
André Platzer

Lecture 14
October 12, 2010

1 Introduction

Several optimizations are easier to perform on SSA form, because SSA needs less analysis. Some optimizations can even be built into SSA construction. Advanced optimizations need advanced static analysis techniques on IR trees and SSA.

Some of what is covered in these notes is explained in [App98, Chapters 17.2–17.3]. A canonical reference on optimizing compilers is the book by [Muc97], which also contains a brief definition of SSA.

2 Constant Folding

The idea behind constant folding is simple but successful. Whenever there is a subexpression that only involves constant operands then we precompute the result of the arithmetic operation during compilation and just use the result. That is, whenever we find a subexpression $n_1 \odot n_2$ for concrete number constants n_1, n_2 and some operator $\odot$, we compute the numerical result n of the expression $n_1 \odot n_2$ and use n instead. Let us say this subexpression occurs in an assignment $x = n_1 \odot n_2$, which, for clarity, we write $x \leftarrow n_1 \odot n_2$ here.

$$\frac{x \leftarrow n_1 \odot n_2 \quad n_1 \odot n_2 = n}{x \leftarrow n}\ CF$$

For that, of course, we have to be careful with the arithmetical semantic of the target language. That is especially for expressions that raise an overflow or division by zero error. Large expressions with constant operands seldom occur in source code. But around 85% of constant folding comes from code generated by address arithmetic. Constant folding also becomes possible after other optimizations have been performed like constant propagation.

More advanced constant folding proceeds across basic blocks and takes SSA ϕ functions into account. For instance, if we have an expression in SSA that, after constant propagation, is of the form $n_1 \odot \phi(n_2, n_3)$ for some operator $\odot$ and numerical constants n_1, n_2, n_3 then we can perform constant folding across ϕ:

$$\frac{x \leftarrow n_1 \odot \phi(n_2, n_3) \quad n_{12} = n_1 \odot n_2 \quad n_{13} = n_1 \odot n_3}{x \leftarrow \phi(n_{12}, n_{13})} \; CF_\phi$$

3 Inverse Operations

Another optimization is to delete inverse operations, e.g., for unary minus:

$$\frac{-(-a)}{a} \; Inv$$

Again, inverse operators seldom occur in source code but may still arise after other optimizations have been used.

4 Common Subexpression Elimination (CSE)

The goal of common subexpression elimination (CSE) is to avoid repetitive computation for subexpressions that occur repeatedly. Common subexpressions occur very frequently in address arithmetic or in generated source code. Studies show, for instance, that 60% of all arithmetic is address arithmetic in PL/1 programs. A source code expression $a.x = a.x{+}1$ for instance yields intermediate code with duplicate address arithmetic

```
t1 = a + offsetx;
t2 = a + offsetx;
*t2 = *t1 + 1;
```

If two operations always yield the same result then they are semantically equivalent. In SSA, if two expressions e, e' are syntactically identical (i.e., the same operators and the same operands) then they are semantically equivalent. Beware, however, that this is not the case in other intermediate representations, where static analysis is still necessary to determine if the same operands still hold the same values or may already hold different ones. In SSA we get this information for free from the property that each location is (statically) only assigned once. Thus, wherever it is available, it holds the same value.

Consequently, in SSA form, for syntactically identical subexpressions e, e' we can remove the computation e' and just store and use the result of e if e dominates e' (otherwise the value may not be available).

Therefore, what we need to do to implement CSE on SSA is the following. For each subexpression e of the form $a \odot b$ for some operator $\odot$ at an SSA node k, we need to find all SSA nodes k' that have the subexpression e. The canonical way to solve this is to maintain a hash table for all expressions and lookup each expression e in it. If we are at node k' with expression e and the hash table tells us that expression e occurs at a node k and k dominates k' then we reuse the value of e from k at k'.

```
for each node k of the form $a \odot b$ do
  look up $a \odot b$ in hash table
  if node j found in hash table that dominates k then
      use result from j instead of $a \odot b$ in k
  else
      leave k as is and put $a \odot b$ into hash table
```

Note that the effect of "use result of j" may depend on the choice of the SSA intermediate representation. For arbitrary SSA representations, the value of an arbitrary subexpression $a \odot b$ may have to be stored into a variable at the dominator j in order to even be accessible at k.

For an SSA representation that only allows one operation for each of the instructions (within a basic block), this is simpler. That is, consider an SSA representation where basic blocks only allows operations of the form $x = a \odot b$ for an operator $\odot$ and variables (but not general expressions) a and b, then only top-level common subexpressions need to be identified and their values will already have been stored in a variable. To illustrate, suppose we are looking at a basic block in which one of the instructions is $y_7 = a \odot b$ for variables a, b then we only need to look for instructions of the form $x = a \odot b$ Thus, if $x_5 = a \odot b$ is one of the instructions in a node j that dominates k, then all we need to do to "use result of j" at k is to replace $y_7 = a \odot b$ in k by $y_7 = x_5$, which, in turn will be eliminated by copy propagation or value numbering.

The CSE algorithm can also be integrated into the SSA construction, because the construction will yield all dominators of k before k (see SSA lecture). Combining CSE with SSA construction also reduces the storage complexity of the SSA graph. Finally, it helps using normalizing transformations like commutativity and distributivity before CSE.

For non-SSA form, we also have to be extra careful that the variables in the subexpression must always still hold the same value. And we need to store the subexpression in a temporary variable, when the other target may possibly be overwritten.

5 Constant propagation

Constant propagation propagates constant values of expressions like $x_2 = 5$ to all dominated occurrences of x_2. That is we just substitute $x_2 = 5$ into all dominated occurrences of x_2. In non-SSA form, extra care needs to be taken that the value of x_2 cannot be different at the occurrence. Constant propagation is somewhat similar to CSE where the operator $\odot$ is just the constant operator (here 5) that does not take any arguments. It is usually implemented either using the CSE hash tables or implicitly during SSA construction.

For non-SSA we also need to be careful that no other definition of x_2 may possibly reach the statement and that the $x_2 = 5$ definition surely reaches the statement without being overwritten (possibly maybe).

6 Copy propagation

Copy propagation propagates values of copies like $x_2 = y_4$ to all dominated occurrences of x_2. That is we just substitute $x_2 = y_4$ into all dominated occurrences of x_2. In non-SSA form, extra care needs to be taken that the value of x_2 cannot be different at the occurrence. Constant propagation is somewhat similar to CSE where the operator $\odot$ is just the identity operator that only takes one argument. It is usually implemented either using the CSE hash tables or implicitly during SSA construction. The register coalescing optimization during register allocation is a form of copy propagation.

For non-SSA we also need to be careful that no other definition of x_2 may possibly reach the statement and that the $x_2 = y_4$ definition surely reaches the statement without being overwritten (possibly maybe) and that no definition of y_4 may possibly reach the statement.

7 Normalization

Normalizing transformations do not optimize the program itself but help subsequent optimizations find more syntactically identical expressions. They use algebraic laws like associativity and distributivity. Their use is generally restricted by definitions of the evaluation order (for Java), by the exception order (Java, Eiffel), or by the limitations of floating-point arithmetic (which is neither associative nor distributive).

A simple normalization is to use commutativity $a + b = b + a$. But we could use this equation in both directions? Which one do we use? If we use it arbitrarily then we may still miss identical expressions. Instead, we fix an order that will always normalize expressions. We first fix an order $\prec$ on all operators. For instance, the order in which we constructed the various expressions during SSA construction. And then we order commutative operations so that the small operand (with respect to the order $\prec$) comes first:

$$\frac{a + b \quad b \prec a}{b + a}\, C_+ \qquad \frac{a * b \quad b \prec a}{b * a}\, C_*$$

How do we use distributivity $a * (b + c) = (a * b) + (a * c)$? Again we need to decide in which direction we use it. This time we have to fix an order on the operators. Let us fix $* \prec +$ and use

$$\frac{a * b + a * c \quad * \prec +}{a * (b + c)}\, D$$

But here we already have to think carefully about overflows. The operation $b + c$ might overflow the data range even if $a * b + a * c$ does not (e.g., for $a = 0$). On an execution architecture where range overflows trigger exceptions, using distributivity might be unsafe. When we strictly stick to modular arithmetic, however, distributivity is safe.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.