Lecture Notes on
Basic Optimization Anlaysis

15-411: Compiler Design
André Platzer

Lecture 15
October 14, 2010

1 Introduction

Several optimizations are easier to perform on SSA form, because SSA needs less analysis. In this lecture we discuss some dataflow analysis techniques that are needed to lift optimizations to non-SSA form. We also discuss postoptimization.

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 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.

3 Reaching Expressions

The optimizations above have been presented for SSA intermediate representations, where syntactical identity is the primary criterion for semantical equivalence of terms and where def-use relations are represented explicitly in the SSA representation. So we are done with those optimizations as far as SSA is concerned.

For non-SSA, this is not so easy, because we explicitly have to compute all required information by static analysis. We have already seen how reaching definitions can be computed in a previous lecture on dataflow analysis.

Reaching expressions analysis is very similar to reaching definitions analysis from the dataflow analysis lecture. The difference is essentially that we do not care so much about the variable in which an expression has been stored, but only if the expression could have been computed before already. We say that the expression $a \odot b$ at $l : x \leftarrow a \odot b$ *reaches* a line l' if there is a path of control flow from l to l' during which no part of $a \odot b$ is redefined. In logical language:

- $\mathsf{reaches}(l, a \odot b, l')$ if the expression $a \odot b$ at l reaches l'.

We only need two inference rules to defines this analysis. The first states that an expression reaches any immediate successor. The second expresses that we can propagate a reaching expression to all successors of a line l' we have already reached, unless this line also defines a part of the expression.

$$\frac{l : x \leftarrow a \odot b \quad \mathsf{succ}(l, l')}{\mathsf{reaches}(l, a \odot b, l')}\ RE_1 \qquad \frac{\mathsf{reaches}(l, a \odot b, l') \quad \mathsf{succ}(l', l'') \quad \neg\mathsf{def}(l', a) \quad \neg\mathsf{def}(l', b)}{\mathsf{reaches}(l, a \odot b, l'')}\ RE_2$$

Reaching expression analysis is only needed for a small subset of all expressions during CSE. Thus, it is usually not performed exhaustively but only selectively as needed for some expressions.

4 Available Expressions

Another analysis that is not obvious except for SSA representations is that of available expressions. Reaching expressions capture the expressions by static analysis that could possibly reach a node. But it is not certain that they will, so we cannot always rely on the expression being available under all circumstances. This is what available expressions analysis captures. Which expressions are available at a point no matter what control path has been taken before.

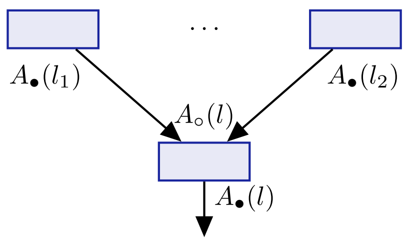
An expression $a \odot b$ is available at a node k if, on every path from the entry to k, the expression $a \odot b$ is computed at least once and no subexpression of $a \odot b$ has been redefined since the last occurrence of $a \odot b$ on the path. There is a crucial difference to all the dataflow analyses that we have seen in class before. For available expressions we are not interested in what information *may* be preserved from one location to another because at least one control path provides it (may analysis). We are interested in what information *must* be preserved on all control paths reaching the location so that we can rely on it being present (must analysis).

Unfortunately, must analysis is a tricky match for logic rules, because that keeps adding information, but we cannot (easily) talk about negations. What we would have to say is something like

$$\frac{\neg\exists l'(\mathsf{succ}(l_0, l) \wedge \neg\mathsf{avail}(l_0, a \odot b))}{\mathsf{avail}(l, a \odot b)}\ ?$$

This can still be expressed with logic rules, but it is much more complicated to use them in the right way. We have to saturate appropriately before we interpret negations.

Alternatively, we use a representation of available expression analysis by dataflow equations. We follow the dataflow schema shown in Figure 1 using the definitions from Table 1.



$$A_\bullet(init) = c$$

$$A_\circ(l) = \bigcap_{l_i \mapsto l} A_\bullet(l_i)$$

$$A_\bullet(l) = (A_\circ(l) \setminus kill(l)) \cup gen(l)$$

Figure 1: Dataflow analysis schema for available expressions

Table 1: Dataflow analysis definitions for available expressions

Statement l	$gen(l)$	$kill(l)$
$init$	$A_\bullet(init) = \emptyset$	
$x \leftarrow a \odot b$	$\{a \odot b\} \setminus kill(l)$	$\{e : e$ contains $x\}$
$x \leftarrow *a$	$\{*a\} \setminus kill(l)$	$\{e : e$ contains $x\}$
$*a \leftarrow b$	$\emptyset$	$\{*z :$ for all $z\}$
goto l'	$\emptyset$	$\emptyset$
if $a > b$ **goto** l'	$\emptyset$	$\emptyset$
l' :	$\emptyset$	$\emptyset$
$x \leftarrow f(p_1, \ldots, p_n)$	$\emptyset$	$\{e : e$ contains x or any $*z\}$

5 Peephole Optimization

One of the simple-most optimizations is peephole optimization by McKeeman from 1965 [McK65]. Peephole optimization is a postoptimization in the backend after/during instruction selection has been performed. It is an entirely local transformation. The basic idea is to move a sliding peephole, usually of size 2, over the instructions and replace this pair of instructions by cheaper instructions. After precomputing cheaper instruction sequences for the set of all pairs of operations, a simple linear sweep through the instructions is used to optimize each pair. The matter is a little more involved in the case of conditional jumps, where both possible target locations need to be considered. Peephole optimization can be quite useful for CISC architectures to glue code coming originally from independent AST expressions or to use fancy address modes. For instance, there are address modes that combine

$$a[i]; i++ \quad \rightsquigarrow \quad a[i++]$$

Typical peephole optimizations include

store R, a; load a, R	$\rightsquigarrow$	store R, a	superfluous load
imul 2, R	$\rightsquigarrow$	ashl 2,R	multiplication by constant
iadd x,R; comp 0,R	$\rightsquigarrow$	iadd x, R	superfluous comparisons
if b then x:=y	$\rightsquigarrow$	t:=b; x:=(t) y	IA-64 predicated assignment

The major complication with peephole optimization is its mutual dependency with other instruction selection optimizations like pipeline optimization. Peephole optimization should be done before instruction selection for pipeline optimization but it cannot be done without instructions having been selected. Other than that, it can lead to globally suboptimal choices, because it is a local optimization. The fact that it is an entirely local transformation, however, makes it easy to implement. Its limit is that it has only a very narrow and local view of the program.

Postoptimizations can be fairly crucial. A strong set of postoptimizations can in fact lead to a lot more than 10% performance improvement. A few compilers only use postoptimizations (e.g., lcc). Unfortunately, postoptimizations are often quite processor dependent and not very systematic.

References

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

[McK65] William M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, 1965.

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