Compilers and "Optimization" Contributions to the FP98 Course

Richard Fateman
Computer Science Division, EECS Dep't
University of California at Berkeley

March 25, 2016

Abstract

This section consists of draft Notes on language and compilers. Here we address the issue of what compilers should be allowed to do, encouraged to do, and what should be forbidden.

1 Prerequisites

We assume that the audience is familiar with the IEEE 754 standard for binary floating-point arithmetic, and has familiarity with at least one high-level numerically-oriented programming language (Fortran or C most likely).

2 Simplification or optimization

Optimization vs. Non-pessimization?

Here are some categories to consider

2.1 Admissible optimizations

Any transformations that consistently and inevitably returns the same answer, except is usually faster. This generally includes mucking about with array index calculations.

2.2 Bad optimizations

There are many mathematical simplifications that just don't always work, and thus *DO NOT WORK*. It is worse than tiresome to deal with compilers that claim they can rearrange floating-point operations to any "mathematically equivalent" expression. It is dangerous.

(A+B)-A is not the same as B. parentheses must be respected. example: (1.0+1.0E-20)-1.0. A+B+C can be evaluated left to right or in some other order. Can we be clear when these are allowed?

2.3 Can Bad be OK?

We can pursue this idea: **Licensing Associativity: at compile time**

If A*B/C can be evaluated in any order, or A+B+C can be evaluated in any order, then this can speed up matrix operations, and can be used to justify the decomposition of loops.

Rationale for rearrangements include: better memory access (cache hits), better use of pipelines (keeping two pipelines full by adding odds together and evens together).

Licensing evaluation precision: Widest, narrowest, fixed (e.g. double).

Licensing Fused MAC: Specifically: use it; use it if the compiler can figure out some way to do it; specifically don't use it.

Warning: Licenses must be done in the program text, not the command line.

2.4 Loop Optimizations

There are a raft of possible changes to code that can be examined for doing violence to floating-point semantics. Some of them are easily illustrated as source-to-source transformations. Others require looking at generated code (lacking an explicit model of what is done in registers, some of these optimizations cannot be expressed as source code).

We should provide an option to provide source-to-source translations results for confirmation by the programmer.

2.4.1 Loops

Moving operations out of a loop may be incorrect, even if they merely replace computations by constants.

```
loop for x in range
   if (condition(x)) then 0.0/0.0   //force exception & trap
           else compute(x)
 end
```

Clearly the 0.0/0.0 is a constant, and hence could be evaluated outside the loop. If it is moved outside the loop, it forces an exception without testing the condition. A moderately clever compiler would try to execute this constant calculation earlier, at compile-time. This would probably remove ANY exceptions at run-time, inserting a NaN.

Any expression that could cause an exception but would not necessarily be evaluated, cannot be moved out of a loop! Here's a trick though..

Loop peeling is when you execute the first iteration of a loop "unoptimized" and then re-use the common components in subsequent iterations. For example,

```
// a and c are loop invariant
for(i=0; i < n; i++)
{ b = a/c;
  <rest of loop>}
```

gets turned into

```
// "peeled" first iteration
i=0
if ~(i<n){
t1 = a/c;       // only perform one divide
b = t1
<rest of loop>
i++

// a and c are loop invariant
for(i=1; i < n; i++)
{
  b = t1;
  <rest of loop>
}
}
```

This optimization preserves the exceptional behavior, but this simplistic version is inadequate in general to preserve behavior if sticky flags are tested and reset in the loop (cf. Borneo admits/yields)

Other ways that "constant expressions" could change:

```
for i from 0 to 4
   rounding_mode := i  // change rounding
    a[i]:=f(x)  // compute constant (?) expression
end
```

Similar problems may occur when apparently common subexpressions are "eliminated" but must maintain their own identities because of mode variations.

2.4.2 Trichotomy

```
a<b or a>b or a=b
```

for any two ordinary numbers. But either a or b being a NaN means they are all false.

```
if (a>b) then x else y   is NOT THE SAME as
if (a<=b) then y else x
```

The kinds of branching and loop exits that are compiled must respect the NaN. Many, if not all, compiler books routinely suggest rearrangement of inequalities and Boolean conditions containing them, based on trichotomy. The tradition continues even today with new books that are in other respects authoritative.

2.5 Equality

The existence of NaNs means that using memory equality for numeric equality is not valid. That is, one cannot assume that a single memory pattern is numerically equal to itself. Programming languages that use pointers to objects (Java, C, Lisp) must generally distinguish

between numerical equality and object isomorphism. Lisp has several predicates: eq, equalp, =.

The existence of signed zeros means that two memory patterns that are different may nevertheless be numerically equal.

What does it mean to compare two numbers of different precision?

Mathematica says: two numbers are equal if they differ in about 2 decimal digits of the precision of the least precise of the two numbers.

There is a folk rule that you should not compare floats for equality. But making it impossible to compare them is unhelpful.

(for example, there is no problem in rapid comparisons of integers that are exactly representable as floats;)

It is hard to know what use can be made from arithmetic that behaves as given below

```
m=SetPrecision[123.0,5]
m1=m+1/10
m3=m+1/10000000000000000


m==m1==m3 is True

m1> m   is True     m3> m  is False
m1>=m   is True     m3>=m  is True
m1==m   is True     m3==m  is True
m1=!=m  is True     m3=!=m is False
m1===m  is False    m3===m is True
m1<=m   is False    m3<=m  is True
m1< m   is False    m3< m  is False
m-m1==0 is False    m-m3==0 is True

m1^2-m^2==24.61   is True
m1^2-m^2==30      is True
m1^2==m^2         is True
```

But it is part of the Mathematica design, based on a bad idea called Significance arithmetic.

2.6 Strength reduction

Do compilers still change

```
for i from 1 to n do
   z= i*x
  ...
end
to
 z=0;
for i from 1 to n do
```

```
  z=z+x
  ..
end ??
```

This provides an opportunity for n rounding errors instead of one. This kind of reduction works quite well for integer operations.