5. Avoiding Overflow in Constant Expression Evaluation

W. M. McKeeman
Digital Equipment Corporation
110 Spit Brook Road
Nashua, NH 03062

Abstract

Evaluation of constant expressions during translation raises the possibility of overflow. A set of checks that are safe against integer overflow is presented. Similar, but less reliable safe checks for floating overflow are also presented.

Introduction

In C translation, both during preprocessing and during semantic analysis, the translator must evaluate constant expressions.[1]. The sequence of arithmetic operations defined by the C grammar can be used for the evaluation. Because of integral promotions and usual arithmetic conversions, the operands are always of the same type.[2, section 3.2] Integer overflow can occur when that type is either int or long and the operator is one of `+ - * / \%'. Floating overflow can occur for floating types and the same operators except `\%'.[2, section 3.3.5] Because of conditional evaluation of operators `&& || ?: ' and nonevaluation of the argument of the sizeof operator, some apparent overflows do not occur.

Furthermore, overflow can occur when converting a constant. Exact constant conversion, particularly for floating types, is a topic of its own. Here is assumed that constant operands have been converted to representable internal form.

When the host and target arithmetic are the same, the compiler can use the operator that it is evaluating directly. For fault detection, the compiler can enable arithmetic traps and issue a diagnostic as part of the recovery. More generally, because the host and target machine arithmetic may be different, the needed operations are constructed out of ones available in the host C. In this case traps may not help. The evaluation problem is to get a value when there is one and to issue a diagnostic when there is not.
Safe Checks

A solution to overflow detection not requiring traps, accounting for host/target differences and conditional evaluation is presented below. The general form of the solution is to replace each operation with an overflow-safe version, and defer diagnostics until the complete expression is evaluated.

All of this detail can be hidden inside a C function

\[
\text{Operand \ Op(Operand left, Operator op, Operand right)};
\]

where the data type Operand is a struct containing a field each for the type, whether the data has a value (failure has not yet occurred), the locator and diagnostic for the leftmost failure (if any) and a union of containers for the target values. Within function Op there is switch logic to separate out the cases.

When overflow occurs, a function is called to construct a diagnostic and mark the result invalid. For example, let \( M \) represent INT_MAX and \( m \) represent INT_MIN. The int operation \( a + b \) overflows when \( a + b > M \) or \( a + b < m \). Neither test can be directly evaluated without causing overflow itself. The form of the overflow-safe C expression for \( a + b \) is

\[
\text{IntAddOverflow}(a,b) \ ? \ \text{Error}(a, b, \text{"int + overflow") : IntAdd(a,b)}
\]

where

- the function \( \text{IntAddOverflow} \) evaluates the safe check expressions for addition in Table 1,
- the function \( \text{Error} \) places a diagnostic in the Operand result and marks it invalid, or
- the correct value is placed in Operand result by function \( \text{IntAdd} \) (simply \( a+b \) if the host and target are the same).

The unsafe checking expressions for integral subtraction, multiplication and division are analogous to that for addition. Unary arithmetic negation \((-a)\) is equivalent to \(0 - a\). Unary plus cannot cause overflow. The safe remainder operator is defined in terms of the other safe operators. This definition is consistent with the C Standard.[2, section 3.3.5]

\[
a \% b = a - b * (a/b)
\]

The C Standard requires that floating constants be converted and, in integral expressions, these constants be immediately cast to integral values. In a file scope initializer for a float variable, the expression must be constant. For example
<table>
<thead>
<tr>
<th>unsafe check</th>
<th>safe check</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a + b &gt; M$</td>
<td>$b &gt; 0 \land a &gt; M - b$</td>
</tr>
<tr>
<td>$a + b &lt; m$</td>
<td>$b &lt; 0 \land a &lt; m - b$</td>
</tr>
<tr>
<td>$a - b &gt; M$</td>
<td>$b &lt; 0 \land a &gt; M + b$</td>
</tr>
<tr>
<td>$a - b &lt; m$</td>
<td>$b &gt; 0 \land a &lt; m + b$</td>
</tr>
<tr>
<td>$\neg b$</td>
<td>$\text{check } 0 - b$</td>
</tr>
<tr>
<td>$a * b &gt; M$</td>
<td>$(b &gt; 1 \land a &gt; M/b) \lor (b &lt; 0 \land a &lt; M/b)$</td>
</tr>
<tr>
<td>$a * b &lt; m$</td>
<td>$(b &gt; 1 \land a &lt; m/b) \lor (b &lt; -1 \land a &gt; m/b)$</td>
</tr>
<tr>
<td>$a/b &gt; M$</td>
<td>$b = 0 \lor (b = -1 \land a = m)$</td>
</tr>
<tr>
<td>$a/b &lt; m$</td>
<td>$b = 0$</td>
</tr>
<tr>
<td>$a % b$</td>
<td>$\text{check } a - b \times (a/b)$</td>
</tr>
</tbody>
</table>

Table 1: Unsafe and Safe Integral Overflow Checks

FLT_MAX-3.0

is an acceptable initializer but the subtraction can be deferred to run time. Table 2 gives checks for floating operators, except that roundoff and underflow are ignored.

For the same reason that

```c
sizeof(LONG_MAX+LONG_MAX)
```

is well defined and causes no overflow,

```c
sizeof(FLT_MAX+FLT_MAX)
```

likewise is well defined and causes no overflow. Similarly,

```
2 || 1 / 0
```

is a constant expression which does not cause evaluation of the division.[2, footnote 55]

In general each safe check is a combination of operations, tests against the limiting values and calls to `Error()`. The safe check is derived by algebraic manipulation of the corresponding unsafe check.

Tables 1 and 2 gives the correspondence between unsafe and safe checks. The `V` and `∧` operators must be evaluated from left to right to avoid unsafe operations. The value $M$ is the positive maximum for each type; the value $m$ is the negative minimum for the integral types. We assume that $m + M \leq 0$.

The derivation of the safe check for integral multiplication follows. The unsafe expression $a * b > M$ needs to be checked when the signs of $a$ and $b$
<table>
<thead>
<tr>
<th>unsafe check</th>
<th>safe check</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a + b &gt; M$</td>
<td>$b &gt; 0.0 \land a &gt; M - b$</td>
</tr>
<tr>
<td>$a + b &lt; -M$</td>
<td>$b &lt; 0.0 \land a &gt; -M - b$</td>
</tr>
<tr>
<td>$+b$</td>
<td>never fails</td>
</tr>
<tr>
<td>$a - b$</td>
<td>check $a + (-b)$</td>
</tr>
<tr>
<td>$-b$</td>
<td>never fails</td>
</tr>
<tr>
<td>$a + b &gt; M \lor a + b &lt; -M$</td>
<td>$abs(b) &gt; 1.0 \land abs(a) &gt; M/abs(b)$</td>
</tr>
<tr>
<td>$a/b &gt; M \lor a/b &lt; -M$</td>
<td>$b = 0.0 \lor (abs(b) &lt; 1.0 \land abs(a) &gt; M + abs(b))$</td>
</tr>
</tbody>
</table>

Table 2: Unsafe and Safe Floating Overflow Checks

are the same. If both are positive then $a > M/b$ suffices. If both are negative $a < M/b$ suffices.

To derive the positive-positive safe check for multiplication, let $M = k \times b + r$
where $0 \leq r < b$. Then, starting at

$a > M/b$

notice that since $r/b = 0$

$a > M/b = (k \times b + r)/b = k$

or

$a > k$.

Since $a$ is at least 1 greater than $k$

$a + b \geq (k + 1) \times b = k \times b + b > k \times b + r = M$

or

$a + b > M$.

as was to be shown. The rest of the integral cases are similar although there is a
tedious case analysis on the signs of the operands.

Summary

A C compiler must deal with the possibility of overflow during the evaluation of
constant expressions. One method is provide overflow-safe operators using an
overflow-safe check prior to the evaluation of the operator. Some safe operators
can be built out of other safe operators. The safe checks are derived by case
analysis and algebraic manipulation of unsafe checks.

References


William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. He can be reached at mckeeman@tle.dec.com.