Single Use Expressions, Optimizing Compilers, and Rounding

1 Single Use Expressions

In general, if we apply straightforward interval computations to estimate the range of a given expression \( y = f(x_1, \ldots, x_n) \) on given intervals \([x_1, x_i]\), we get not the exact range \( y \), but an enclosure \( Y \supset y \) for the range.

For example, for the expression \( f(x) = x^2 - 2x + 1 = x \cdot x - 2x + 1 \) on the interval \([1, 2]\), straightforward interval computations lead to

\[
[1, 2] \cdot [1, 2] - 2 \cdot [1, 2] + 1 = [1, 4] - [2, 4] + 1 = [1 - 4, 4 - 2] + 1 = [-3, 2] + 1 = [1, 3],
\]

while the actual range of the expression \( f(x) = (x - 1)^2 \) on this interval is equal to \([0, 1]\).

However, if we have an expression in which each variable occurs only once — such expressions are called Single Use expressions (SUE, for short), then straightforward interval computations lead to

\[
([1, 2] - 1)^2 = [0, 1]^2 = [1, 2],
\]

2 Optimizing Compilers

What is an optimizing compiler. Compilers try to simplify the expression to make them computable faster. Let us give two examples.

- An optimizing compiler will replace the expression \( a \cdot b + a \cdot c \) that requires two multiplications and one addition with an equivalent expression \( a \cdot (b+c) \) that requires one multiplication and one addition.

- An optimizing compiler will replace the expression

\[
\frac{1}{1 + \frac{a}{b}}
\]
that requires two divisions and one addition with an equivalent expression

\[
\frac{b}{a+b}
\]

that requires one division and one addition.

**Should we rely on optimizing compilers?** How does the use of optimizing compilers affect interval computations?

**Sometimes, an optimizing compiler helps.** Let us assume that we want to estimate the range of the expression \(a \cdot b + a \cdot c\) when \(a \in [-1, 1]\), \(b \in [1, 2]\), and \(c \in [-2, -1]\).

In this example, straightforward interval computations lead to

\[
[-1, 1] \cdot [1, 2] + [-2, -1] \cdot [-2, 2] = [-4, 4].
\]

For this example, optimizing compiler – aiming to minimize the number of multiplications – will transform the original expression into \(a \cdot (b + c)\). The new expression is SUE, so straightforward interval computations lead to the exact range:

\[
[-1, 1] \cdot ([1, 2] + [-2, -1]) = [-1, 1] \cdot [-1, 1] = [-1, 1].
\]

**Sometimes, an optimizing compiler makes things worse.** Let us consider the problem of estimating the expression

\[
\frac{1}{1 + \frac{a}{b}}
\]

when \(a = b = [10, 20]\). This is a SUE expression, so straightforward interval computations lead to

\[
\frac{1}{1 + \frac{[10, 20]}{[10, 20]}} = \frac{1}{1 + [10, 20]} \cdot \frac{1}{[10, 20]} = \frac{1}{1 + [0.5, 2]} = \frac{1}{[1.5, 3]} = [0.333 \ldots, 0.666 \ldots].
\]

For this example, an optimizing compiler – aiming to minimize number of divisions – will transform the original expression into

\[
\frac{b}{b + a}
\]

For this new expression, straightforward interval computations lead to a wider interval:

\[
\frac{[1, 2]}{[1, 2] + [1, 2]} = [1, 2] \cdot \frac{1}{[2, 4]} = [1, 2] \cdot [0.25, 0.5] = [0.25, 1].
\]
3 Rounding

**Reminder.** When we multiply two numbers with fixed number of digits after a decimal point in a computer, we get more digits than a computer can represent. So, the computer usually rounds this result to the nearest.

For example, if we only use numbers with 2 digits after the decimal point, and a computer needs to multiply 0.13 and 0.93, it gets the value $0.13 \cdot 0.23 = 0.1203$ which it then rounds to 0.12.

**Problem.** This will not work if we are trying, e.g., to find the guaranteed bound for the product $y = x_1 \cdot x_2$ when $x_1 \in [0.10, 0.13]$ and $x_2 \in [0.90, 0.93]$. The actual range of the product is

$$[y, \bar{y}] = [0.10 \cdot 0.90, 0.13 \cdot 0.93] = [0.09, 0.1203].$$

This means that we can guarantee that if:

- $x_1$ is in the interval $[0.10, 0.13]$ and
- $x_2$ is in the interval $[0.90, 0.93]$,

then we their product $y = x_1 \cdot x_2$ is:

- larger than or equal to 0.09 and
- smaller than or equal to 0.1203.

However, if we use the usual roundings and get the interval $[0.09, 0.12]$, we can no longer guarantee that the product is smaller than or equal to 0.12 – the product could be equal to 0.1203 and thus, be larger than 0.12.

**Solution.** To get guaranteed bounds, we need to round down the lower endpoint, and to round up the upper endpoint.

**Example.** In the above example, when the actual range is $[0.09, 0.1203]$, this means that after proper rounding, we get the interval $[0.09, 0.13]$. 

3