Bounds-Consistent Propagation for Integer Multiplication

1 Difference from Existing Propagator

The previous propagator for multiplication in Pumpkin was not bounds-consistent. Sign handling was only partially implemented and was weak, it was only able to infer whether $c \geq 0$ or $c \leq 0$ and for a or b whether they were $a, b \geq 1$ or $a, b \leq -1$.

$$a \in [2, 10], \quad b \in [2, 10], \quad c \in [5, 100]$$

When checked manually, it can be seen that $c \geq 2 \cdot 2 = 4$, so $c_{\min} = 5$ survives. However, there does not exist any a, b in $[2, 10] \times [2, 10]$ such that $a \times b = 5$ since the only way to get 5 is from integers is 1×5 which is outside the domain. The smallest achievable product is $2 \times 3 = 6$. A bounds-consistent propagator must tighten $c_{\min}$ to 6.

Also it wasn't able to infer anything about c when a or b crossed 0. For example it would skip propagating a, b to c if $a \in [x_1, x_2]$ where $x_1 < 0 \wedge x_2 > 0$.

2 New Approach

There are 3 new propagation logic steps implemented to ensure bound-consistency.

2.1 Step 1: Multiplication Arithmetic

Propagating c from a and b. This step is the more trivial step. In order to propagate from a, b to c, there are 4 unique values which c might be bounded by to consider sign handling. Hence:

$$c_{\min} \geq \min(a_{\min}b_{\min},\ a_{\min}b_{\max},\ a_{\max}b_{\min},\ a_{\max}b_{\max})$$

$$c_{\max} \leq \max(a_{\min}b_{\min},\ a_{\min}b_{\max},\ a_{\max}b_{\min},\ a_{\max}b_{\max})$$

By doing this, propagation can be made even if 0 is crossed in the boundaries of a or b

2.2 Step 2: Division Arithmetic

Propagating a from c and b (and symmetrically for b). This is less trivial because both sign handling and rounding correctness needs to be implemented correctly. Because the propagation works on integers and division can result in floats, correct flooring/ceiling needs to be chosen. When we are able to bound a by saying it is $a \leq x$ where $x \in \mathbb{Q}$, we can say it is $a \leq \lfloor x \rfloor$. When it is possible to bound a by $a \geq x$ it is possible to say $a \geq \lceil x \rceil$. Hence the `ceil_div` and `floor_div` functions are needed.

When b does not contain 0, the bounds of a are obtained by dividing the four combinations of c and b bounds using the rounding arithmetic explained above.

When b's interval contains 0 but c's does not, division by the endpoints alone can over-prune. Values near zero ($b = \pm 1$ in this case because this propagator works using integers) can produce extreme quotients that the endpoint-only calculation misses. Hence this extra calculation/check is necessary to ensure bounds consistency alongside the four combination check.

When both b and c contain 0, no useful bound on a can be derived: $0 \times x = 0$ for $\forall x \in \mathbb{Z}$, so the propagation is skipped in this scenario.

The logic is exactly the same when propagating b from a and c, only variable renaming is needed.

2.3 Step 3: Support Checking

After step 1 and 2, the bounds are tighter, it might not be (and most times isn't) enough to achieve bounds consistency. The formal definition of the bounds consistency is:

$$p(\mathcal{D}) = \mathcal{D} \implies \forall x_i \in X,\ (\exists \theta_{LB} \in \mathrm{Sol}(C) : \theta_{LB}(x_i) = LB(x_i)) \wedge (\exists \theta_{UB} \in \mathrm{Sol}(C) : \theta_{UB}(x_i) = UB(x_i))\,.$$

In simple terms, for every lower bound and upper bound, there needs to exist an actual solution in the sets of other variables. This is achieved through a linear scan inwards until a supported value is found:

- For a bound x of a: check whether there exists $b' \in [b_{\min}, b_{\max}]$ such that $x \cdot b' \in [c_{\min}, c_{\max}]$.
- For a bound x of c: check whether there exist $a' \in [a_{\min}, a_{\max}]$ and $b' \in [b_{\min}, b_{\max}]$ with $a' \cdot b' = x$. This is done by iterating over the smaller of the two factor intervals and testing divisibility.

If there does not exist any supported value for a variable, the propagator reports a conflict.

3 Examples

Example 1: Tightening on unsupported bounds. For: $a \in [2, 10]$, $b \in [2, 10]$, $c \in [5, 100]$ basic interval arithmetic is not enough to make this bounds consistent. It does not tighten c (since $2 \times 2 = 4 < 5$). Then support check catches that 5 cannot be obtained in $[2, 10] \times [2, 10]$ so it prunes it even tighter, so $c_{\min} = 6$ (since $2 \times 3 = 6$).

Example 2: No possible support. For: $a \in [2, 3]$, $b \in [2, 3]$, $c = 5$. It is not possible to prune this tighter using interval multiplication/division logic because $2 \times 2 \leq 5 \leq 3 \times 3$. However, it is impossible to find any support for any bound. So the propagator checks all values from 2 and tightens sequentially, when it reaches that $c_{min} > 5$ it reports a correct conflict.

Example 3: Negative domains. $a \in [-5, -2]$, $b \in [2, 5]$, $c \in [-10, 24]$. Interval arithmetic gives $c \in [\min(-10, -25, -4, -10), \max(-10, -25, -4, -10)] = [-25, -4]$, which does not tighten $c_{\min}$ since $-10 > -25$. But it does tighten $c_{\max}$ since $24 > -4$ so the bound becomes $c \in [-10, -4]$. The support checks show that bounds are supported, hence the final correct result for c is $c \in [-10, -4]$.

Example 4: Mixed signs with zero-straddling factors. $a \in [-4, 7]$, $b \in [-3, 5]$, $c \in [11, 30]$. Since b crosses zero, division-based logic on a produces very wide bounds (dividing by $b = \pm 1$ gives bounds up to ± 30) and does not tighten a or b. The support check finds that $c = 11$ has no factorisation within $[-4, 7] \times [-3, 5]$ (11 is prime; $11/1 = 11 \notin [-4, 7]$ and $-11/(-1) = 11 \notin [-4, 7]$). The next value, $12 = (-4) \times (-3)$, is supported. Hence the final result becomes $a \in [-4, 7]$, $b \in [-3, 5]$, $c \in [12, 30]$.

4 Conclusion and Limitations

The implementation includes 13 unit tests covering many cases. All tests pass, along with clippy and formatting checks. So the propagator works as intended and is bounds-consistent. However, making it a bounds-consistent propagator comes with downsides. There are 2 different linear scans needed to ensure bounds consistency logic which is computationally costly, especially for some edge cases. For example if it tries to find support for a very large prime number, it would take $O(n)$ time to find out that it is not supported. In practise, this does not happen often because the arithmetic logic bounds the values aggressively, eliminating most bad edge cases. Hence the main limitation is this implementation may be inefficient on very large inputs.