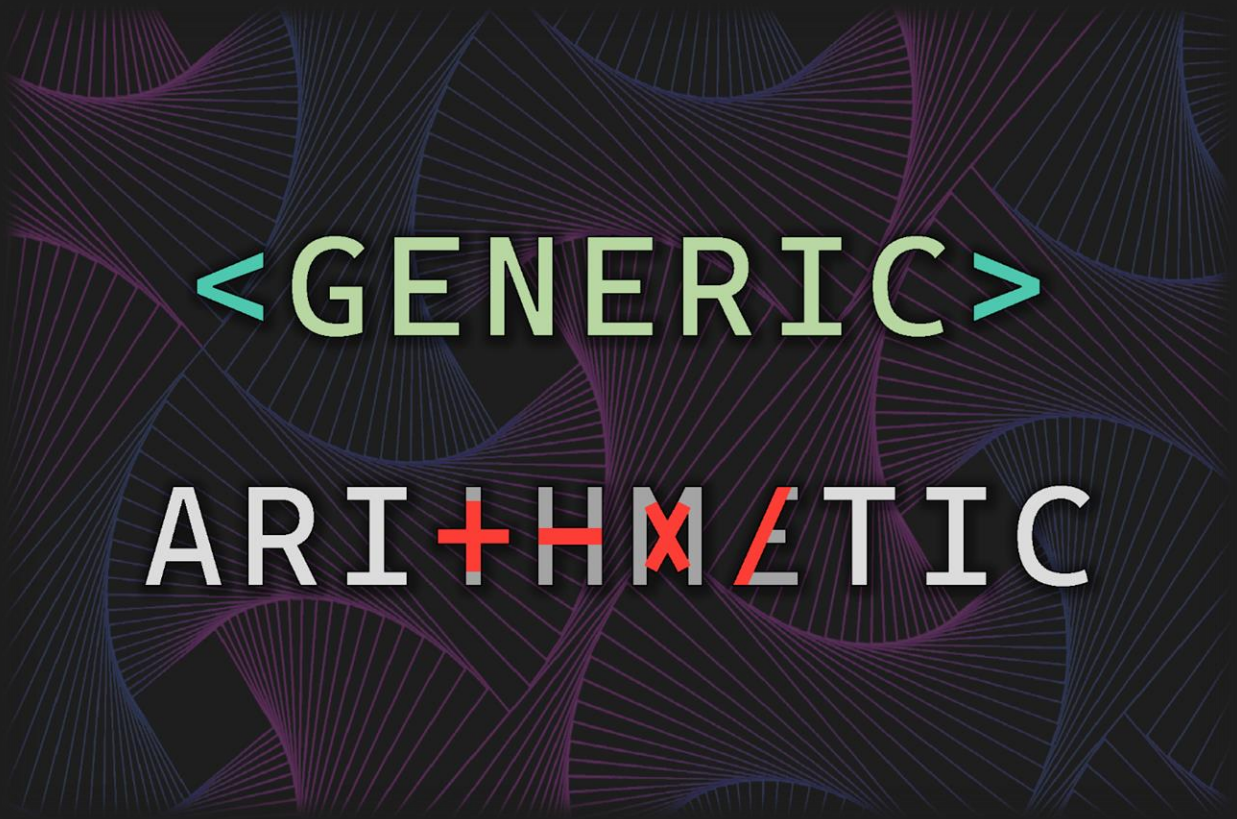

<GENERIC>
ARI+HM/TIC

Contents

Basic Usage

Computable

The **Computable<T>** struct is the simplest way to do calculation in a template context.

You can instantiate it using the **new** keyword with a value of type **T**, or using the static **From** method with a value of another type. You can then use it just like you would a non-generic value.

It will implicitly cast to a **T**, you can also access its **value** field.

For example, let's consider this method:

```
public static T Lerp_DoesNotCompile<T>(T a, T b, T t)
{
    if (t < 0)
        t = 0;
    else if (t > 1)
        t = 1;

    return a + ((b - a) * t);
}
```

We can use the **Computable<T>** struct to make it compile like this:

```
public static T Lerp_Computable<T>(T a, T b, T t)
{
    Computable<T> cA = new Computable<T>(a);
    Computable<T> cB = new Computable<T>(b);
    Computable<T> cT = new Computable<T>(t);

    Computable<T> zero = Computable<T>.From(0);
    Computable<T> one = Computable<T>.From(1);

    if (cT < zero)
        cT = zero;
    else if (cT > one)
        cT = one;

    return cA + ((cB - cA) * cT);
}
```

Calculate and Evaluate

If you need to work with multiple template types, you can use the **Calculate<TResult>** and **Evaluate** static classes.

The former provides **static methods** equivalent to **arithmetic operators** like +, -, ++, &, etc. The latter provides **static methods** equivalent to **comparison operators** like ==, >, etc.

For example, let's use these static methods to make the above method compile:

```
public static T Lerp_StaticMethods<T>(T a, T b, T t)
{
    if (Evaluate.LessThan(a, 0))
        t = Calculate<T>.Cast(0);
    else if (Evaluate.GreaterThan(a, 1))
        t = Calculate<T>.Cast(1);

    return Calculate<T>.Addition(a,
                Calculate<T>.Multiply(t,
                    Calculate<T>.Subtract(b, a)));
}
```

Default Computators

Configuration

The computator classes oversee the different generic operations you can do.

You can choose the default behavior by editing the **configuration asset** (*top bar > Tools > Generic Arithmetic > Configuration*).

Different default computator types exist, suitable for different scenarios.

Types

Expression

Generate optimized code at runtime using **reflection** and **Linq.Expression**. The first call of any operation will be slow but subsequent calls will be very performant.

Dynamic

Uses the **Dynamic Language Runtime** to compute operation at runtime. This is constant but not very good performance wise.

Throw

Always throw, use this if you want to make sure you never rely on a default computator.

IL2CPP

All default computator won't work when compiling with **IL2CPP**, which is necessary on some target platforms.

If you use **IL2CPP**, you should use **Throw** as your default computator type, and create manual computators as needed (see below). Source generation will come in a later update!

Otherwise, you should probably use **Expression** as your default computator type because it is more performant.

Manual Computators

Usage

Manual computators are classes defined as **scripts in the Assets folder**. They will be used over default computators.

They are 4 main cases where you should consider making a manual computator:

- To support a type when compiling with **IL2CPP**.
- For better performances than default computators.
- To resolve some operator ambiguities manually.
- To extend the arithmetic capabilities of a type.

To make one, all you must do is create a script with a class that derives from some closed dedicated generic classes (see below). The template parameters will define when to use this computator.

It will be integrated automatically. You can verify that your class is correctly detected by checking the **configuration** asset.

Classes

TrioCalculator<TResult, TParamA, TParamB>

Define all binary arithmetic operators but left and right shift.

DuoCalculator<TResult, TParam>

Define all unary arithmetic operators, cast, left and right shift.

DuoEvaluator<TParamA, TParamB>

Define all binary comparison operators.

SoloEvaluator<TParam>

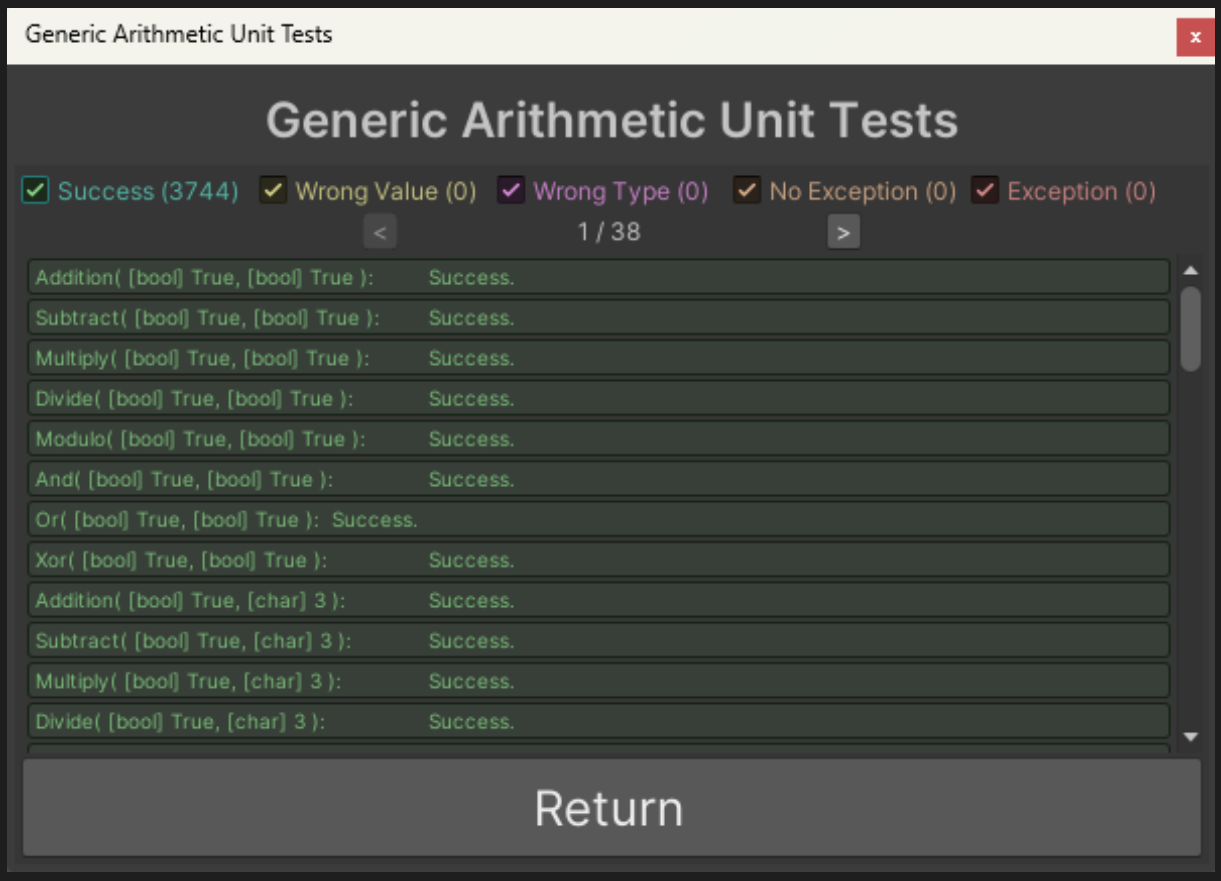
Define the true and false operators.

Other

Unit Tests Window

This was used during development to make sure the **Expression default computators** are operating exactly like the **CLR** would.

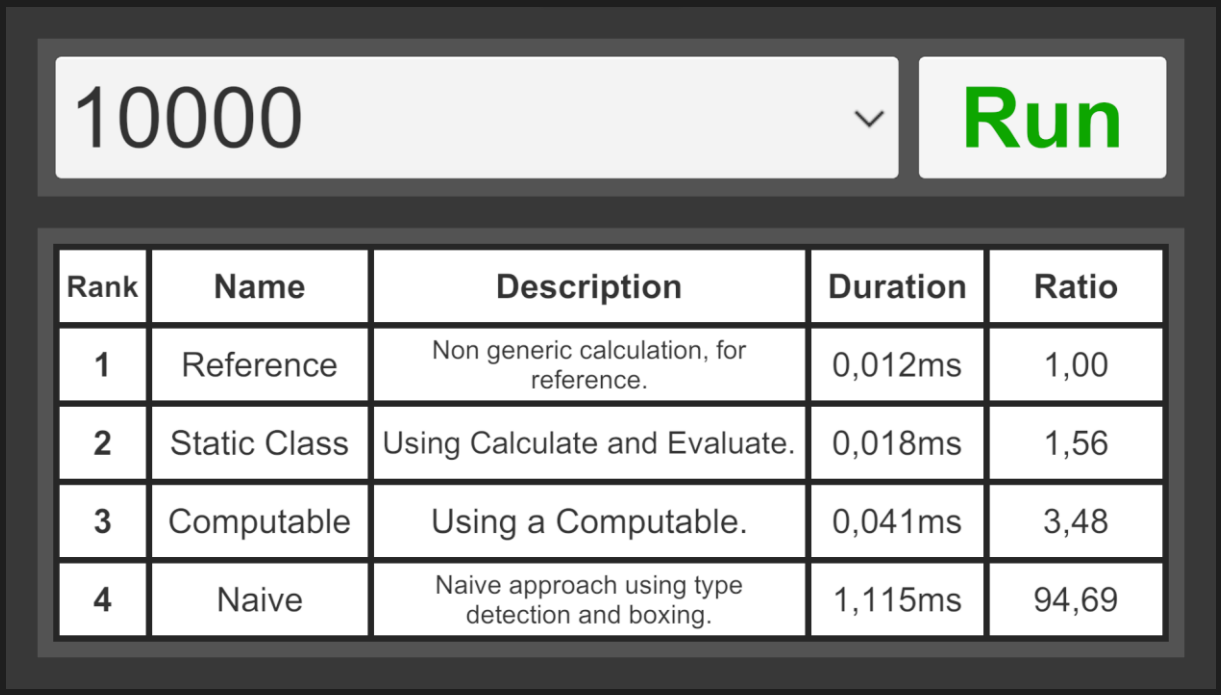
You can use it too in case you want to verify that the tool is working correctly with a specific type, or if you want to make custom changes to the scripts (*top bar > Tools > Generic Arithmetic > Unit Tests*).



Benchmark Demo

The demo is a simple benchmark (speed test). It compares 4 different ways to handle generic operations.

Use it to get an idea of how fast the tool is depending on how you configured it.



Rank	Name	Description	Duration	Ratio
1	Reference	Non generic calculation, for reference.	0,012ms	1,00
2	Static Class	Using Calculate and Evaluate.	0,018ms	1,56
3	Computable	Using a Computable.	0,041ms	3,48
4	Naive	Naive approach using type detection and boxing.	1,115ms	94,69

Final words

Thank you for getting this tool!

If you have any questions, discover a bug, or have a feature idea, please don't hesitate to contact me at justetools@gmail.com.

If you enjoy **Generic Arithmetic**, please consider leaving a review. Your feedback helps improve the tool!