Artículo
Dmitrii Kuznetsov · 4 feb, 2020 Lectura de 16 min

DataOps: Let's learn how to work with physical objects in IRIS

When do implicit assumptions in code and arithmetic operators pose a real danger, and how can this danger be overcome?

Constantinople in the 5th century AD

(It was possible for people to build grandiose buildings and vehicles without the use of computers and robots 1,600 years ago.)

If you assign dimensionless numbers to physical parameters in your code, you risk misunderstandings and errors when calculating and converting between units of measurement. This applies to both composite units of measurement, such as those for energy, pressure, and power, as well as to the simple basic units for time, length, temperature, and so on. You also need to remember to include the ability to convert between metric system (SI) units and American customary units.

> The most infamous errors in programs related to the incorrect processing of physical units have led to the failure of entire projects:
>
> 1. NASA in 1998: The Mars Climate Orbiter satellite, which included equipment that was developed by scientists from the United States, England, and Russia and cost $125 million, was lost because a subcontractor responsible for engineering tasks failed to convert pound-force to metric newtons. As a result

of the error, after a 286-day trip at high speed, the satellite entered the Martian atmosphere, where its communications systems failed due to the resulting overloads. The device ended up one hundred kilometers below the planned orbit and 25 km below the altitude at which it would still have been possible to correct the situation. As a result, the satellite crashed. https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

2. The US Air Force in 2007: Twelve F-22 Raptors (fifth-generation fighter aircraft in service with the United States Air Force), costing $140 million each, were dispatched on their first international flight to Okinawa. Everything went fine until the squadron crossed the date line, on the western side of which the date shifts one day forward relative to the eastern side. After crossing this line, all 12 fighters simultaneously issued an error message essentially equivalent to the blue screen of death.

The aircraft lost access to data on the remaining amount of fuel, speed and altitude sensors, and the planes also partially lost communications. For several hours, America ’ s most modern fighter jets flew across the ocean completely helpless. In the end, the planes were only able to land thanks to the skill of their pilots.

So, what caused the problem? The designers from Lockheed Martin didn ’ t even consider the possibility that the plane would cross the date line. It never occurred to them that somewhere they might need to add or subtract one day.

3. Roscosmos (the Russian space agency) in 2018: The Frigate upper stage of the Soyuz-2.1b launch vehicle with 19 satellites performed its first orbital insertion maneuver in the wrong orientation. At the time when the upper stage separated from the launch vehicle, the rotation error was 363 degrees. The Frigate started to reverse the rotation, but it simply wasn ’ t able to make the correction in time, since the main propulsion motor switched on in accordance with the launch sequence and it provided impulse while the stage was incorrectly oriented. The upper stage started to fall towards the ocean. As a result, the maneuver didn ’ t provide an accelerating boost as intended, but rather it acted as a brake. Instead of inserting itself into its exchange orbit, the upper stage together with its payload fell to the Earth and sank in the Atlantic Ocean.

Driver, pay attention!
These signs may be describing either two totally different speed limits or the same one!
Remember the rules and conventions. Though both signs say "50", these are two different speeds.

And even today, when we are already 20% through the twenty-first century, when we advocate for the adoption of strict type systems and compliance with a theory of categories, and a theory of types is one of the major priorities for programmers, the most popular programming languages still don ’ t support physical quantities. Indeed, there ’ s no general agreement on how to implement such standardization uniformly.

Do we have even a trivial understanding of how physical quantities can be applied in programming languages?

Physical numbers can be processed using the typical arithmetic operations: addition, subtraction, multiplication, division, exponentiation, and so on. Moreover, both the order of these operations and the significant difference in the results of different operations matter. For example, when adding two lengths, the unit will be preserved. And when multiplying two lengths, you get the area. Let ’ s try to resolve these metamorphoses.

There are only seven basic units of measurement in the SI system: the meter, kilogram, second, Kelvin, ampere, candela, and mole. The remaining units are derived from these basic units.

Let's consider for example, what ’ s probably the most famous formula:

$E=m\bullet c^2$

Or it can be expressed using units of measurement:

$J=kg\bullet(m/s)^2$

The joule can be represented by various equivalent combinations of physical quantities depending on the subject area:

$J = kg \cdot m^2 / s^2 = N \cdot m = W \cdot s = C \cdot V$

It should be mentioned that in addition to SI units, there are also units for currencies, angles, geocoordinates, typographic and screen quantities, and many more that have been invented over the course of many centuries.

Let's put these aside for the moment, however, and focus instead on the "real" physical units.

Let's start with one well-known headache that programmers encounter: seconds. What problem could be easier than this? But, remember, not so long ago people had to confront the computer apocalypse relating to the change of the millennium: the “ Y2K problem. ” A similar problem visible on the horizon will occur on January 19, 2038, when

32 bits will no longer be enough:

01111111111111111111111111111111

It ’ s probably the case that each programming language has its own way of counting seconds. Here are just a few examples from various languages:

- C: The time function reads the current system clock in seconds. The value of the system clock is the time in seconds that has elapsed since 12:00 AM on January 1, 1970. Fear of the year 2038 in all its glory.
- Java: The time is calculated as the number of milliseconds that have elapsed since midnight on January 1, 1970. Again we run into the 2038 trap. There are the variants java.util.Date and java.util.Calendar, which have their own peculiarities. And there ’ s also a Date Time API. This is an entire zoo of implementations!
- Rust: Take a look at the Chrono library — here everything has been pretty well and interestingly thought out.
- Python: Time intervals are floating point numbers that have been calculated in seconds. Special points in time are expressed as the number of seconds that have elapsed since 12:00 AM on January 1, 1970 (epochs). Python has a time module that provides functions for working with time and for converting between various representations. The time.time function returns the current system time in ticks as calculated since 12:00 AM on January 1, 1970. Do we face the ill-fated number of 2038 in this case as well?
- ObjectScript: Time is calculated from 12:00 AM on December 31, 1840. The number of days and seconds since then are calculated separately. The format in which time is saved is ddddd, sssss. The last date that this scheme supports is December 31, 9999. Earlier dates are expressed in the SQL Julian date format, which provides 100% protection against both Y2K and 2038 problems.

What does ISO 8601 (the international standard for date- and time-related data) tell us? Consider this:

2019-07-20T22:32:42/P3Y6M4DT12H30M17S describes a period of time of 3 years, 6 months, 4 days, 12 hours, 30 minutes, and 17 seconds starting from 10:32:42 PM on July 20, 2019.

Conclusion: ISO 8601 helps us to avoid the 2038 error, but it still doesn ’ t save us. We need a data structure in which to save time information. Simple cavalry attacks using INT, DOUBLE, or STRING aren ’ t sufficient by themselves, though in light of ISO 8601 STRING seems to be good!

We ’ ll leave years, months, weeks, and days for a later time. Right now, let's confine our consideration of scale to seconds, including milliseconds and microseconds. (Rust even has picoseconds and nanoseconds.) This suggests another important factor to take into account: the scale factor. And this exists for all physical quantities, whether at the nano, micro, milli, kilo, mega, or tera scales. All of these scales are separated from each other by powers of 10.

There ’ s a certain fundamental principle that comprises:

- Seven base units (in the SI system).
- Dimensionless units/factors.
- Derived units with their own names.
- Alternative names for derived units for various subject areas.
- A scale factor provided in the IEEE-754 format for floating-point numbers.
- A comparison of the SI unit system with other systems of units of measurement.
- Different languages with their own designations and names for units.

  The whole modern world has switched to the metric system of units (Le Système International d'Unités, or SI), with only three countries in the world (USA, Myanmar, and Liberia) that ignore this convention. Not only does this not change the main question, it makes it even more pressing. How can we express the values of physical units with their units of measurement in programming languages while making the most of type systems and corresponding checks when converting and executing code?

Let's Create a DSL

Formulating physical data in a program is akin to constructing a special physical language. In this case, a good solution would be to use a suitable programming language, such as Julia or Kotlin.

The following are examples of the physical nature and ambiguity of numerical values in code:

- Measuring time, time and date, which we already described earlier.
- Font sizes as well as screen and web page dimensions.
- Physical quantities, which is the topic that we are considering.
- Money, currencies, and exchange rates.

The use of numbers in a program without taking into account the units of measurement is equivalent to a default value that ' s hidden in the head of a developer. Even if the context of the number is described in the comments or the documentation, this isn ' t real code that can be executed by a computer. There ' s no way to use the power of a translator and runtime to find errors and inconsistencies in the used data types.

A Story from the Ada language

Back in 1985, Narein Jehani, who authored many studies and books on the programming languages Ada, Pascal, and C, made an interesting observation when describing derived types in Ada. I will present his examples and comments briefly without losing the main idea.

```
"new" is used to create a derived type:

type LENGTH is new FLOAT;

type AREA is new FLOAT;

The three dimensions of a parallelepiped:

L, W, H: LENGTH;

A: AREA;

V: VOLUME;
```

The use of derived types allows the compiler to automatically detect errors in semantically meaningless operations:

```
L + A

A + V

A := V;
```

However, it also interprets errors as semantically meaningful (and correct) operations:

```
L * A

A := L * W;
```

And it doesn ' t reveal any semantically meaningless operations, such as:

```
L := W * H
```

In retrospect, Jehani regretted that no type system for "units of measurement" was built into the Ada language. As he wrote, a simpler (and, in the author ’ s opinion, better) approach to the mechanism of derived types is the "units of measurement" approach, which represents them as objects. This might take the following form in Ada:

```
L, W, H: FLOAT {feet}

A: FLOAT {square feet}

V: FLOAT {cubic feet}
```

You can read more in N.H. Gehani, "Ada ’ s Derived Types and Units of Measure, Software," Practice and Experience, Vol. 15(6), 1985.

A Story from C++

Much more recently, in 2012, Bjarne Stroustrup, the creator of the C++ programming language, spoke about the updates that had been introduced in the 11th version of the language. One of the topics that he considered important to describe more clearly was the need to focus on interfaces and avoid the use of primitive data types, like integers and real numbers, in order to prevent obvious errors. (See https://parasol.tamu.edu/people/bs/622-GP/C++11-style.pdf and

https://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style for more information.)

So, instead of:

```
void increase_speed(double)

Object obj; ... obj.draw()

Rectangle(int,int,int,int)
```

It ’ s better to use:

```
void increase_speed(Speed)

Shape& s; ... s.draw()

Rectangle(Point top_left, Point bottom_right)

Rectangle(Point top_left, Box_hw b)
```

Then, just as with Ada, the topic of physical units pops up. Did you notice the Speed type in the example above? It was possible to implement the Speed type before, but it was very cumbersome, so it fell out of favor with developers:

```
template<int M, int K, int S>

    struct Unit { // a unit in the MKS system
    enum { m=M, kg=K, s=S };

};

template<typename Unit> // a magnitude with a unit

    struct Value {

        double val; // the magnitude

        explicit Value(double d) : val(d) {} // construct a Value from a double

};

using Speed = Value<Unit<1,0,-1>>; // meters/second type

Speed sp1 = Value<1,0,0> (100)/ Value<0,0,1> (9.8); // very explicit

Speed sp1 = Value<M> (100)/ Value<S> (9.8); // use a shorthand notation

Speed sp1 = Meters(100)/Seconds(9.8); // abbreviate further still

And a simple addition was made to C++: user suffix literals.

using Second = Unit<0,0,1>; // unit: sec

using Second2 = Unit<0,0,2>; // unit: second*second

constexpr Value<Second> operator"" s(long double d) // a f-p literal suffixed by ‘s’

{

    return Value<Second> (d);

}

constexpr Value<Second2> operator"" s2(long double d) // a f-
p literal suffixed by ‘s2’

{

    return Value<Second2> (d);

}
```

Now the text of the program looks quite elegant, and the compiler is also able to read it for purposes of error tracking:

```
Speed sp1 = 100m/9.8s;       // very fast for a human

Speed sp2 = 100m/9.8s2;      // error (m/s2 is acceleration)

Speed sp3 = 100/9.8s;        // error (speed is m/s and 100 has no unit)

Acceleration acc = sp1/0.5s; // too fast for a human
```

The State of Affairs in Python

Python is widely used as an accessible language for calculations, so it ’ s not surprising that it includes several ready-made packages that make it easier to work with units. See the review article "Quantities and Units in Python ” by Gastón Hillar and Gaston Hillar, September 10, 2013.

In the same vein, consider the F# language. Of course, this language isn ’ t as popular as C++, Java, or Python, but it pays special attention to the issue of how to work with physical data. Everything that ’ s needed for working with units of measurement is built directly into the syntax and semantics of the language:

```
[<Measure>] type ft

let gravityOnEarth = 32.2<ft/s^2>

let heightOfBuilding = 130.0<ft>

let speedOfImpact = sqrt (2.0 * gravityOnEarth * heightOfBuilding)
```

See http://typesatwork.imm.dtu.dk/material/TaWPaperTypesAtWorkKennedy.pdf and

https://docs.microsoft.com/dotnet/fsharp/language-reference/units-of-measure for more information.

Let's Give ObjectScript a Try

What can be done in a language with dynamic type systems as well as an impeccable innate ability to store any data?

Let ’ s calculate how long it takes to travel from the surface of the Earth to the International Space Station (ISS), which orbits at an altitude of 410 km and at an authorized speed of 50 km/h.

Step 0: The π theorem

To begin with, let's go down the beaten path and look at the π theorem, which is the solution incorporated into C ++. For example:

[x] = Ll * Mm * Tt * Θθ * Ii * Jj * Nn

or

[l, m, t, θ, i, j, n]

That is, any units of measurement from our base list of SI units can be represented by a simple degree matrix in the same order. For example, for velocities (m/s), where m^1 and c^-1, this will be:

[0, 1, -1, 0, 0, 0, 0]

This is very similar to the declaration of the dynamic array %Library.DynamicObject in ObjectScript:

set speed = [0, 1, -1, 0, 0, 0, 0]

See also:

- Dimensional analysis
- π theorem
- System of physical quantities

Step 1: Recognizing/identifying literals

Numeric literals in ObjectScript aren ’ t especially helpful for us. They aren ’ t yet supported in the language, but we ’ ll use string literals to the fullest extent possible:

```
set literal = "50 ?m/h»
```

Or even like this:

```
set literal = "50 mph»
```

We can break it down into two components: the velocity value and the units of measurement. Let the separator in our makeshift DSL be a space:

```
set literalValue = $PIECE(literal, " ", 1)

set value = $DOUBLE(literalValue)

set literalUnit = $PIECE(literal, " ", 2)

set literalUnitNumerator = $PIECE(literalUnit, "/", 1)

set literalUnitDenominator = $PIECE(literalUnit, "/", 2)
```

Let's single out the scale factor (if any), the units that exist outside the system, and the elementary components from the SI system in the units of measurement:

```
set scale = $case($EXTRACT(literalUnitNumerator),

    "k": 3,

    "M": 6,

    "G": 9,

    "m": -3,

    "μ": -6,

    "n": -9,

       : 0

)

set value = value * (10 ** scale)

if (scale '= 0) set literalUnitNumerator = $EXTRACT(literalUnitNumerator, 2, *)
```

To achieve greater accuracy, we need to take into account that the usual joules, pascals or watts are integral units, and we must take care to convert them to their "normal" form:

```
set unit = $case(unit,

    "l": "kg*m^2/s^2",

    "Pa": "kg/m*s^2",

    "W":  "kg*m^2/s^3",

)
```

Or we could express them using matrix notation:

```
if unit = "J" { set unitSI = [2, 1, -2, 0, 0, 0, 0] } // kg*m^2/s^2

elseif unit = "Pa" { set unitSI = [-1, 1, -2, 0, 0, 0, 0] } // kg/m*s^2

elseif unit = "W":  { set unitSI = [2, 1, -3, 0, 0, 0, 0] } // kg*m^2/s^3
```

And we convert the elementary units of measurement and their degrees into matrix form, for example:

```
set speed = [0, 1, -1, 0, 0, 0, 0]
```

Step 2: Let's learn the operators +, -, *, and /

While we ’ re not able to reload arithmetic operators in ObjectScript, we can solve this problem using blunt force methods: we can write the missing functions Add, Sub, Mul, and Div.

The logic of how they work is the same, with the exception of our most important factor: all operands must be checked for compatibility. In this case, the units of measurement for the result should be reduced to the corresponding form.

For example, it ’ s quite simple for addition and subtraction, because the units of the operands and parameters must coincide with each other, and the result will have the same units:

`Add()`

Usage:

```
set result = oper1.Add(oper2)
```

For multiplication and division (and exponentiation, too, by the way), the units of the operands and parameters can be completely different, and the resulting units of measurement are obtained by adding the matrices:

`Mul()`

Usage:

```
set result = oper1.Mul(oper2)
```

As a result, we get a new operand object that always has the correct units of measurement or an error message. This is what we sought to achieve in our statement of the problem.

All that ’ s left is to prepare the physical machine data in a form that can be understood by people.

Step 3: Formatting for export

Since we saved not only the value together with the system units of measurement, but also the original units of measurement, there ’ s a lot that we can still do. For example:

```
Class Speed.Base Extends %RegisteredObject

{

Property literal As %String;

Property value As %Double;

Property unitType As array Of %Integer;

Property unitBaseString As %String;



Method Value() As %Double

{

    if (..value = "") do ..Parse()

    return ..value

}



Method Unit() As %String

{

    if (..unitBaseString = "") do ..Parse()

    return ..unitBaseString

}



Method ToString(format = "SI", scale = "") As %String

{

    if ((format = "SI") && (scale = "")) set literal = ..Value() _" "_ ..Unit()

    if (format = "nonSI") set literal = ..Value()/1609.34 _" mph"
    if (scale = "k") set literal = ..Value()/1000 _" "_"k"_..Unit()

    return literal

}


}
```

```
set valueWithUnit = ##class(Speed.Base).%New()

set valueWithUnit.literal = "3660 km/h"
```

- Output the result in a single SI system format

```
w valueWithUnit.ToString()
```

- Convert the units of measurement to their original or a compatible format

- `w valueWithUnit.literal`

- Convert the presentation from elementary SI units to units that are more conventional for a particular area of application

```
w valueWithUnit.ToString("notSI")
```

- And we can even add scale factors.

```
w valueWithUnit.ToString(, "k")
```

Conclusions

So, the following are my proposals concerning units of measurement and what I ’ d like to see implemented in ObjectScript:

1. The use of physical units of measurement as part of the system of language types makes sense in order to reduce the number of calculation errors, and would make it more convenient for programmers to explicitly indicate familiar notations, including when transmitting, storing, and presenting such data.
2. The language should support the ability to reload operators and use literal suffixes. This would greatly simplify the task of developing application class libraries and strengthening the type system through the use of physical quantities with error detection at the code compilation stage.
3. It ’ s possible to keep everything the way it is now and to solve this problem at a higher level of abstraction. For example, you could delegate the processing of physical data to AI algorithms and machine learning.

If you have suggestions about how physical data can be correctly supported in ObjectScript or, what would be even better, about how physical types can be implemented, please add your comments. Together we can implement an API with a class system or a library for such data in IRIS, to the envy of other languages!

The main thing is that, in contrast to the conventions and descriptions that are hidden in the comments, such "documentation" that ’ s integrated into the code will never become out-of-date, since it ’ s checked every time the source code passes through the compiler and the program is executed.

#Principiante #Compilador #Modelo de datos #ObjectScript #InterSystems IRIS