Conversion and Promotion

Julia has a system for promoting arguments of mathematical operators to a common type, which has been mentioned in various other sections, including Integers and Floating-Point Numbers, Mathematical Operations and Elementary Functions, Types, and Methods. In this section, we explain how this promotion system works, as well as how to extend it to new types and apply it to functions besides built-in mathematical operators. Traditionally, programming languages fall into two camps with respect to promotion of arithmetic arguments:

- Automatic promotion for built-in arithmetic types and operators. In most languages, built-in numeric types, when used as operands to arithmetic operators with infix syntax, such as `+`, `-`, `*`, and `/`, are automatically promoted to a common type to produce the expected results. C, Java, Perl, and Python, to name a few, all correctly compute the sum `1 + 1.5` as the floating-point value `2.5`, even though one of the operands to `+` is an integer. These systems are convenient and designed carefully enough that they are generally all-but-invisible to the programmer: hardly anyone consciously thinks of this promotion taking place when writing such an expression, but compilers and interpreters must perform conversion before addition since integers and floating-point values cannot be added as-is. Complex rules for such automatic conversions are thus inevitably part of specifications and implementations for such languages.
- No automatic promotion. This camp includes Ada and ML – very "strict" statically typed languages. In these languages, every conversion must be explicitly specified by the programmer. Thus, the example expression `1 + 1.5` would be a compilation error in both Ada and ML. Instead one must write `real(1) + 1.5`, explicitly converting the integer `1` to a floating-point value before performing addition. Explicit conversion everywhere is so inconvenient, however, that even Ada has some degree of automatic conversion: integer literals are promoted to the expected integer type automatically, and floating-point literals are similarly promoted to appropriate floating-point types.

In a sense, Julia falls into the "no automatic promotion" category: mathematical operators are just functions with special syntax, and the arguments of functions are never automatically converted. However, one may observe that applying mathematical operations to a wide variety of mixed argument types is just an extreme case of polymorphic multiple dispatch – something which Julia's dispatch and type systems are particularly well-suited to handle. "Automatic" promotion of mathematical operands simply emerges as a special application: Julia comes with pre-defined catch-all dispatch rules for mathematical operators, invoked when no specific implementation exists for some combination of operand types. These catch-all rules first promote all operands to a common type using user-definable promotion rules, and then invoke a specialized implementation of the operator in question for the resulting values, now of the same type. User-defined types can easily participate in this promotion system by defining methods for conversion to and from other types, and providing a handful of promotion rules defining what types they should promote to when mixed with other types.

Conversion

The standard way to obtain a value of a certain type `T` is to call the type's constructor, `T(x)`. However, there are cases where it's convenient to convert a value from one type to another without the programmer asking for it explicitly. One example is assigning a value into an array: if `A` is a `Vector{Float64}`, the expression `A[1] = 2` should work by automatically converting the `2` from `Int` to `Float64`, and storing the result in the array. This is done via the `convert` function.

The `convert` function generally takes two arguments: the first is a type object and the second is a value to convert to that type. The returned value is the value converted to an instance of given type. The simplest way to understand this function is to see it in action:

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> convert(UInt8, x)
0x0c

julia> typeof(ans)
UInt8

julia> convert(AbstractFloat, x)
12.0

julia> typeof(ans)
Float64

julia> a = Any[1 2 3; 4 5 6]
2×3 Array{Any,2}:
 1  2  3
 4  5  6

julia> convert(Array{Float64}, a)
2×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
```

Conversion isn't always possible, in which case a `MethodError` is thrown indicating that `convert` doesn't know how to perform the requested conversion:

```
julia> convert(AbstractFloat, "foo")
ERROR: MethodError: Cannot `convert` an object of type String to an object of type A
[...]
```

Some languages consider parsing strings as numbers or formatting numbers as strings to be conversions (many dynamic languages will even perform conversion for you automatically). This is not the case in Julia. Even though some strings can be parsed as numbers, most strings are not valid representations of numbers, and only a very limited subset of them are. Therefore in Julia the dedicated `parse` function must be used to perform this operation, making it more explicit.

When is `convert` called?

The following language constructs call `convert`:

- Assigning to an array converts to the array's element type.
- Assigning to a field of an object converts to the declared type of the field.
- Constructing an object with `new` converts to the object's declared field types.
- Assigning to a variable with a declared type (e.g. `local x::T`) converts to that type.
- A function with a declared return type converts its return value to that type.
- Passing a value to `ccall` converts it to the corresponding argument type.

Conversion vs. Construction

Note that the behavior of `convert(T, x)` appears to be nearly identical to `T(x)`. Indeed, it usually is. However, there is a key semantic difference: since `convert` can be called implicitly, its methods are restricted to cases that are considered "safe" or "unsurprising". `convert` will only convert between types that represent the same basic kind of thing (e.g. different representations of numbers, or different string encodings). It is also usually lossless; converting a value to a different type and back again should result in the exact same value.

There are four general kinds of cases where constructors differ from `convert`:

Constructors for types unrelated to their arguments

Some constructors don't implement the concept of "conversion". For example, `Timer(2)` creates a 2-second timer, which is not really a "conversion" from an integer to a timer.

Mutable collections

`convert(T, x)` is expected to return the original `x` if `x` is already of type `T`. In contrast, if `T` is a mutable collection type then `T(x)` should always make a new collection (copying elements from `x`).

Wrapper types

For some types which "wrap" other values, the constructor may wrap its argument inside a new object even if it is already of the requested type. For example `Some(x)` wraps `x` to indicate that a value is present (in a context where the result might be a `Some` or `nothing`). However, `x` itself might be the object `Some(y)`, in which case the result is `Some(Some(y))`, with two levels of wrapping. `convert(Some, x)`, on the other hand, would just return `x` since it is already a `Some`.

Constructors that don't return instances of their own type

In *very rare* cases it might make sense for the constructor `T(x)` to return an object not of type `T`. This could happen if a wrapper type is its own inverse (e.g. `Flip(Flip(x)) === x`), or to support an old calling syntax for backwards compatibility when a library is restructured. But `convert(T, x)` should always return a value of type `T`.

Defining New Conversions

When defining a new type, initially all ways of creating it should be defined as constructors. If it becomes clear that implicit conversion would be useful, and that some constructors meet the above "safety" criteria, then `convert` methods can be added. These methods are typically quite simple, as they only need to call the appropriate constructor. Such a definition might look like this:

```
convert(::Type{MyType}, x) = MyType(x)
```

The type of the first argument of this method is a singleton type, `Type{MyType}`, the only instance of which is `MyType`. Thus, this method is only invoked when the first argument is the type value `MyType`. Notice the syntax used for the first argument: the argument name is omitted prior to the `::` symbol, and only the type is given. This is the syntax in Julia for a function argument whose type is specified but whose value does not need to be referenced by name. In this example, since the type is a singleton, we already know its value without referring to an argument name.

All instances of some abstract types are by default considered "sufficiently similar" that a universal `convert` definition is provided in Julia Base. For example, this definition states that it's valid to `convert` any `Number` type to any other by calling a 1-argument constructor:

```
convert(::Type{T}, x::Number) where {T<:Number} = T(x)
```

This means that new `Number` types only need to define constructors, since this definition will handle `convert` for them. An identity conversion is also provided to handle the case where the argument is already of the requested type:

```
convert(::Type{T}, x::T) where {T<:Number} = x
```

Similar definitions exist for `AbstractString`, `AbstractArray`, and `AbstractDict`.

Promotion

Promotion refers to converting values of mixed types to a single common type. Although it is not strictly necessary, it is generally implied that the common type to which the values are converted can faithfully represent all of the original values. In this sense, the term "promotion" is appropriate since the values are converted to a "greater" type – i.e. one which can represent all of the input values in a single common type. It is important, however, not to confuse this with object-oriented (structural) super-typing, or Julia's notion of abstract super-types: promotion has nothing to do with the type hierarchy, and everything to do with converting between alternate representations. For instance, although every `Int32` value can also be represented as a `Float64` value, `Int32` is not a subtype of `Float64`.

Promotion to a common "greater" type is performed in Julia by the `promote` function, which takes any number of arguments, and returns a tuple of the same number of values, converted to a common type, or throws an exception if promotion is not possible. The most common use case for promotion is to convert numeric arguments to a common type:

```
julia> promote(1, 2.5)
(1.0, 2.5)

julia> promote(1, 2.5, 3)
(1.0, 2.5, 3.0)

julia> promote(2, 3//4)
(2//1, 3//4)

julia> promote(1, 2.5, 3, 3//4)
(1.0, 2.5, 3.0, 0.75)

julia> promote(1.5, im)
(1.5 + 0.0im, 0.0 + 1.0im)

julia> promote(1 + 2im, 3//4)
(1//1 + 2//1*im, 3//4 + 0//1*im)
```

Floating-point values are promoted to the largest of the floating-point argument types. Integer values are promoted to the larger of either the native machine word size or the largest integer argument type. Mixtures of integers and floating-point values are promoted to a floating-point type big enough to hold all the values. Integers mixed with rationals are promoted to rationals. Rationals mixed with floats are promoted to floats. Complex values mixed with real values are promoted to the appropriate kind of complex value.

That is really all there is to using promotions. The rest is just a matter of clever application, the most typical "clever" application being the definition of catch-all methods for numeric operations like the arithmetic operators `+`, `-`, `*` and `/`. Here are some of the catch-all method definitions given in `promotion.jl`:

```
+(x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)
```

These method definitions say that in the absence of more specific rules for adding, subtracting, multiplying and dividing pairs of numeric values, promote the values to a common type and then try again. That's all there is to it: nowhere else does one ever need to worry about promotion to a common numeric type for arithmetic operations – it just happens automatically. There are definitions of catch-all promotion methods for a number of other arithmetic and mathematical functions in `promotion.jl`, but beyond that, there are hardly any calls to `promote` required in Julia Base. The most common usages of `promote` occur in outer constructors methods, provided for convenience, to allow constructor calls with mixed types to delegate to an inner type with fields promoted to an appropriate common type. For example, recall that `rational.jl` provides the following outer constructor method:

```
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
```

This allows calls like the following to work:

```
julia> Rational(Int8(15),Int32(-5))
-3//1

julia> typeof(ans)
Rational{Int32}
```

For most user-defined types, it is better practice to require programmers to supply the expected types to constructor functions explicitly, but sometimes, especially for numeric problems, it can be convenient to do promotion automatically.

Defining Promotion Rules

Although one could, in principle, define methods for the `promote` function directly, this would require many redundant definitions for all possible permutations of argument types. Instead, the behavior of `promote` is defined in terms of an auxiliary function called `promote_rule`, which one can provide methods for. The `promote_rule` function takes a pair of type objects and returns another type object, such that instances of the argument types will be promoted to the returned type. Thus, by defining the rule:

```
promote_rule(::Type{Float64}, ::Type{Float32}) = Float64
```

one declares that when 64-bit and 32-bit floating-point values are promoted together, they should be promoted to 64-bit floating-point. The promotion type does not need to be one of the argument types. For example, the following promotion rules both occur in Julia Base:

```
promote_rule(::Type{BigInt}, ::Type{Float64}) = BigFloat
promote_rule(::Type{BigInt}, ::Type{Int8}) = BigInt
```

In the latter case, the result type is `BigInt` since `BigInt` is the only type large enough to hold integers for arbitrary-precision integer arithmetic. Also note that one does not need to define both `promote_rule(::Type{A}, ::Type{B})` and `promote_rule(::Type{B}, ::Type{A})` – the symmetry is implied by the way `promote_rule` is used in the promotion process.

The `promote_rule` function is used as a building block to define a second function called `promote_type`, which, given any number of type objects, returns the common type to which those values, as arguments to `promote` should be promoted. Thus, if one wants to know, in absence of actual values, what type a collection of values of certain types would promote to, one can use `promote_type`:

```
julia> promote_type(Int8, Int64)
Int64
```

Internally, `promote_type` is used inside of `promote` to determine what type argument values should be converted to for promotion. It can, however, be useful in its own right. The curious reader can read the code in `promotion.jl`, which defines the complete promotion mechanism in about 35 lines.

Case Study: Rational Promotions

Finally, we finish off our ongoing case study of Julia's rational number type, which makes relatively sophisticated use of the promotion mechanism with the following promotion rules:

```
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:Integer} = Rationa
promote_rule(::Type{Rational{T}}, ::Type{Rational{S}}) where {T<:Integer,S<:Integer}
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:AbstractFloat} = p
```

The first rule says that promoting a rational number with any other integer type promotes to a rational type whose numerator/denominator type is the result of promotion of its numerator/denominator type with the other integer type. The second rule applies the same logic to two different types of rational numbers, resulting in a rational of the promotion of their respective numerator/denominator types. The third and final rule dictates that promoting a rational with a float results in the same type as promoting the numerator/denominator type with the float.

This small handful of promotion rules, together with the type's constructors and the default `convert` method for numbers, are sufficient to make rational numbers interoperate completely naturally with all of Julia's other numeric types – integers, floating-point numbers, and complex numbers. By providing appropriate conversion methods and promotion rules in the same manner, any user-defined numeric type can interoperate just as naturally with Julia's predefined numerics.

« Constructors Interfaces »