DispatchDesigns

May 16, 2018

1 Multiple Dispatch Designs: Duck Typing, Hierarchies and Traits

1.1 Introduction to Dispatch Designs

Julia is built around types. Software architectures in Julia are built around good use of the type system. This makes it easy to build generic code which works over a large range of types and gets good performance. The result is high-performance code that has many features. In fact, with generic typing, your code may have more features than you know of! The purpose of this tutorial is to introduce the multiple dispatch designs that allow this to happen.

1.2 Duck Typing

If it quacks like a duck, it might as well be a duck. This is the idea of defining an object by the way that it acts. This idea is central to type-based designs: **abstract types are defined by how they act**. For example, a `Number` is some type that can do things like `+`,`-`,`*`, and `/`. In this category we have things like `Float64` and `Int32`. An `AbstractFloat` is some floating point number, and so it should have a dispatch of `eps(T)` that gives its machine epsilon. An `AbstractArray` is a type that can be indexed like `A[i]`. An `AbstractArray` may be mutable, meaning it can be "set": `A[i]=v`.

These abstract types then have actions which abstract from their underlying implmentation. `A.*B` does element-wise multiplication, and in many cases it does not matter what kind of array this is done on. The default is `Array` which is a contiguous array on the CPU, but this action is common amongst `AbstractArray` types. If a user has a `DistributedArray` (`DArray`), then `A.*B` will work on multiple nodes of a cluster. If the user uses a `GPUArray`, then `A.*B` will be performed on the GPU. Thus, if you don't restrict the usage of your algorithm to `Array`, then your algorithm actually is "just works" as many different algorithms.

This is all well and good, but this would not be worthwhile if it was not performant. Thankfully, Julia has an answer to this. Every function auto-specializes on the types which it is given. Thus if you look at something like:

```
In [1]: my_square(x) = x^2
```

```
Out[1]: my_square (generic function with 1 method)
```

then we see that this function will be efficient for the types that we give it. Looking at the generated code:

```
In [2]: @code_llvm my_square(1)
```

```
define i64 @julia_my_square_72669(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}
```

```
In [3]: @code_llvm my_square(1.0)
```

```
define double @julia_my_square_72684(double) #0 {
top:
  %1 = fmul double %0, %0
  ret double %1
}
```

See that the function which is generated by the compiler is different in each case. The first specifically is an integer multiplication `x*x` of the input `x`. The other is a floating point multiplication `x*x` of the input `x`. But this means that it does not matter what kind of `Number` we put in here: this function will work as long as `*` is defined, and it will be efficient by Julia's multiple dispatch design.

Thus we don't need to restrict the type in order to get performance. That means that

```
In [4]: my_restricted_square(x::Int) = x^2
```

```
Out[4]: my_restricted_square (generic function with 1 method)
```

is no more efficient than the version above, and actually generates the same exact compiled code:

```
In [5]: @code_llvm my_restricted_square(1)
```

```
define i64 @julia_my_restricted_square_72686(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}
```

Thus we can write generic and efficient code by leaving our functions unrestricted. This is the practice of duck-typing functions. We just let them work on any input types. If the type has the correct actions, the function will "just work". If it does not have the correct actions, for our example above say `*` is undefined, then a `MethodError` saying the action is not defined will be thrown.

We can be slightly more conservative by restricting to abstract types. For example:

```
In [6]: my_number_restricted_square(x::Number) = x^2
```

```
Out[6]: my_number_restricted_square (generic function with 1 method)
```

will allow any `Number`. There are things which can square which aren't `Number`s for which this will now throw an error (a matrix is a simple example). But, this can let us clearly define the interface for our package/script/code. Using these assertions, we can then dispatch differently for different type classes. For example:

```
In [7]: my_number_restricted_square(x::AbstractArray) = (println(x);x.^2)

Out[7]: my_number_restricted_square (generic function with 2 methods)
```

Now, `my_number_restricted_square` calculates `x^2` on a `Number`, and for an array it will print the array and calculate `x^2` element-wise. Thus we are controlling behavior with broad strokes using classes of types and their associated actions.

1.3 Type Hierarchies

This idea of control leads to type hierarchies. In object-oriented programming languages, you sort objects by their implementation. Fields, the pieces of data that an object holds, are what is inherited.

There is an inherent limitation to that kind of thinking when looking to achieve good performance. In many cases, you don't need as much data to do an action. A good example of this is the range type, for example `1:10`.

```
In [8]: a = 1:10

Out[8]: 1:10
```

This type is an abstract array:

```
In [9]: typeof(a) <: AbstractArray

Out[9]: true
```

It has actions like an `Array`

```
In [10]: a[2]

Out[10]: 2
```

However, it is not an `Array`. In fact, it's just two numbers. We can see this by looking at its fields:

```
In [11]: fieldnames(a)

Out[11]: 2-element Array{Symbol,1}:
          :start
          :stop
```

It is an `immutable` type which just holds the start and stop values. This means that its indexing, `A[i]`, is just a function. What's nice about this is that means that no array is ever created. Creating large arrays can be a costly action:

```
In [12]: @time collect(1:10000000)

  0.038615 seconds (308 allocations: 76.312 MB, 45.16% gc time)
```

But creating an immutable type of two numbers is essentially free, no matter what those two numbers are:

```
In [13]: @time 1:10000000

  0.000001 seconds (5 allocations: 192 bytes)


Out[13]: 1:10000000
```

The array takes $\mathcal{O}(n)$ memory to store its values while this type is $\mathcal{O}(1)$, using a constant 192 bytes (if the start and stop are `Int64`). Yet, in cases where we just want to index values, they act exactly the same.

Another nice example is the `UniformScaling` operator, which acts like an identity matrix without forming an identity matrix.

```
In [14]: println(I[10,10])
         println(I[10,2])

1
0
```

This can calculate expressions like `A-b*I` without ever forming the matrix `eye(n)` which would take $\mathcal{O}(n^2)$ memory.

This means that a lot of efficiency can be gained by generalizing our algorithms to allow for generic typing and organization around actions. This means that, while in an object-oriented programming language you group by implementation details, in typed-dispatch programming you group by actions. `Number` is an abstract type for "things which act like numbers, i.e. do things like $*$", while `AbstractArray` is for "things which index and sometimes set".

This is the key idea to keep in mind when building type hierarchies: things which subtype are inheriting behavior. You should setup your `abstract` types to mean the existance or non-existance of some behavior. For example:

```
In [15]: abstract type AbstractPerson end
         abstract type AbstractStudent <: AbstractPerson end
         abstract type AbstractTeacher <: AbstractPerson end

         mutable struct Person <: AbstractPerson
           name::String
         end

         mutable struct Student <: AbstractStudent
           name::String
```

```
  grade::Int
  hobby::String
end

mutable struct MusicStudent <: AbstractStudent
  grade::Int
end

mutable struct Teacher <: AbstractTeacher
  name::String
  grade::Int
end
```

This can be interpreted as follows. At the top we have `AbstractPerson`. Our interface here is "a Person is someone who has a name which can be gotten by `get_name`".

```
In [16]: get_name(x::AbstractPerson) = x.name

Out[16]: get_name (generic function with 1 method)
```

Thus codes which are written for an `AbstractPerson` can "know" (by our informal declaration of the interface) that `get_name` will "just work" for its subtypes. However, notice that `MusicStudent` doesn't have a `name` field. This is because `MusicStudents` just want to be named whatever the trendiest band is, so we can just replace the usage of the field by the action:

```
In [17]: get_name(x::MusicStudent) = "Justin Bieber"

Out[17]: get_name (generic function with 2 methods)
```

In this way, we can use `get_name` to get the name, and how it was implemented (whether it's pulling something that had to be stored from memory, or if it's something magically known in advance) does not matter. We can keep refining this: an `AbstractStudent` has a `get_hobby`, but a `MusicStudent`'s hobby is always `Music`, so there's not reason to store that data in the type and instead just have its actions implicitly "know" this. In non-trivial examples (like the range and `UniformScaling` above), this distinction by action and abstraction away from the actual implementation of the types allows for full optimization of generic codes.

1.4 Small Functions and Constant Propagation

The next question to ask is, does storing information in functions and actions affect performance? The answer is yes, and in favor of the function approach! To see this, let's see what happens when we use these functions. To make it simpler, let's use a boolean function. Teachers are old and don't like music, while students do like music. But generally people like music. This means that:

```
In [18]: likes_music(x::AbstractTeacher) = false
         likes_music(x::AbstractStudent) = true
         likes_music(x::AbstractPerson) = true

Out[18]: likes_music (generic function with 3 methods)
```

Now how many records would these people buy at a record store? If they don't like music, they will buy zero records. If they like music, then they will pick up a random number between 1 and 10. If they are a student, they will then double that (impulsive Millenials!).

```
In [19]: function number_of_records(x::AbstractPerson)
             if !likes_music(x)
               return 0
             end
             num_records = rand(10)
             if typeof(x) <: AbstractStudent
               return 2num_records
             else
               return num_records
             end
         end

Out[19]: number_of_records (generic function with 1 method)
```

Let's check the code that is created:

```
In [ ]: x = Teacher("Randy",11)
        println(number_of_records(x))
        @code_llvm number_of_records(x)
```

on v0.6, we get:

```
; Function Attrs: uwtable
define i64 @julia_number_of_records_63848(i8** dereferenceable(16)) #0 !dbg !5 {
top:
  ret i64 0
}
```

Notice that the entire function compiled away! Then for a music student

```
x = MusicStudent(10)
@code_typed number_of_records(x)

# Output
CodeInfo(:(begin
        NewvarNode(:(num_records))
        goto 4 # line 30:
        4:  # line 32:
        $(Expr(:inbounds, false))
        # meta: location random.jl rand 279
        # meta: location random.jl rand 278
        # meta: location random.jl rand 367
        # meta: location random.jl rand 370
        SSAValue(1) = $(Expr(:foreigncall, :(:jl_alloc_array_1d), Array{Float64,1},
        # meta: pop location
```

```
        # meta: pop location
        # meta: pop location
        # meta: pop location
        $(Expr(:inbounds, :pop))
        num_records = $(Expr(:invoke, MethodInstance for rand!(::MersenneTwister, :
        (MusicStudent <: Main.AbstractStudent)::Bool # line 34:
        return $(Expr(:invoke, MethodInstance for *(::Int64, ::Array{Float64,1}), :
    end))=>Array{Float64,1}
```

we get a multiplication by 2, while for a regular person,

```
x = Person("Miguel")
@code_typed number_of_records(x)

# Output
CodeInfo(:(begin
        NewvarNode(:(num_records))
        goto 4 # line 30:
        4:  # line 32:
        $(Expr(:inbounds, false))
        # meta: location random.jl rand 279
        # meta: location random.jl rand 278
        # meta: location random.jl rand 367
        # meta: location random.jl rand 370
        SSAValue(1) = $(Expr(:foreigncall, :(:jl_alloc_array_1d), Array{Float64,1},
        # meta: pop location
        # meta: pop location
        # meta: pop location
        # meta: pop location
        $(Expr(:inbounds, :pop))
        num_records = $(Expr(:invoke, MethodInstance for rand!(::MersenneTwister, :
        (Person <: Main.AbstractStudent)::Bool
        goto 22 # line 34:
        22:  # line 36:
        return num_records
    end))=>Array{Float64,1}
```

we do not get a multiplication by 2.

But the key thing to see from the typed code is that the "branches" (the `if` statements) all compiled away. Since types are known at compile time (remember, functions specialize on types), the dispatch of `likes_music` is known at compile-time. But this means, since the result is directly inferred from the dispatch, the boolean value `true/false` is known at compile time. This means that the compiler can directly infer the answer to all of these checks, and will use this information to skip them at runtime.

This is the distinction between compile-time information and runtime information. At compile-time, what is known is:

1) The types of the inputs

2) Any types which can be inferred from the input types (via type-stability)

3) The function dispatches that will be internally called (from types which have been inferred)

Note that what cannot be inferred by the compiler is the information in fields. Information in fields is strictly runtime information. This is easy to see since there is no way for the compiler to know that person's name was "Miguel": that's ephemeral and part of the type instance we just created.

Thus by putting our information into our functions and dispatches, we are actually giving the compiler more information to perform more optimizations. Therefore using this "action-based design", we are actually giving the compiler leeway to perform many extra optimizations on our code as long as we define our interfaces by the actions that are used. Of course, at the "very bottom" our algorithms have to use the fields of the types, but the full interface can then be built up using a simple set of functions which in many cases with replace runtime data with constants.

1.5 Traits and THTT

What we just saw is a "trait". Traits are compile-time designations about types which are distinct from their abstract hierarchy. `likes_music` is a trait which designates which people like music, and it could in cases not be defined using the abstract types. For example, we can, using dispatch, create a `WeirdStudent` which does not like music, and that will still be compile-time information which is fully optimized. This means that these small functions which have constant return values allow for compile-time inheritance of behavior, and these traits don't have to be tied to abstract types (all of our examples were on `AbstracPerson`, but we could've said a `GPUArray` likes music if we felt like it!). Traits are multiple-inheritance for type systems.

Traits can be more refined than just `true/false`. This can be done by having the return be a type itself. For example, we can create music genre types:

```
In [23]: abstract type MusicGenres end
         abstract type RockGenre <: MusicGenres end
         struct ClassicRock <: RockGenre end
         struct AltRock <: RockGenre end
         struct Classical <: MusicGenres end
```

These "simple types" are known as singleton types. This means that we can have traits like:

```
In [24]: favorite_genre(x::AbstractPerson) = ClassicRock()
         favorite_genre(x::MusicStudent) = Classical()
         favorite_genre(x::AbstractTeacher) = AltRock()

Out[24]: favorite_genre (generic function with 3 methods)
```

This means that we can do checks like `if typeof(favorite_genre(x)) <: RockGenre`, and this information will all compile away.

This gives us all of the tools we need to compile the most efficient code, and structure our code around types/actions/dispatch to get high performance out. The last thing we need is syntactic sugar. Since traits are compile-time information, the compiler could in theory dispatch on them. While this is currently not part of Julia, it's scheduled to be part of a future version of Julia (2.0?). The design for this (since Julia is written in Julia!) is known as the Tim Holy Trait Trick (THTT),

named after its inventor. It's described in detail on this page. But in the end, macros can make this easier. A package which implements trait-dispatch is SimpleTraits.jl, which allows you to dispatch on a trait `IsNice` like:

```
In [ ]: @traitfn ft(x::::IsNice) = "Very nice!"
        @traitfn ft(x::::(!IsNice)) = "Not so nice!"
```

1.6 Composition vs Inheritance

The last remark that is needed is a discussion of composition vs inheritance. While the previous discussions have all explained why "information not in fields" makes structural relations compile-time information and increases the efficiency. However, there are cases where we want to share runtime structure. Thus the great debate of composition vs inheritance, comes up.

Composition vs inheritance isn't a Julia issue, it's a long debate in object-oriented programming. The idea is that, inheritance is inherently (pun-inteded) inflexible. It forces an "is a" relation: A inherits from B means A is a B, and adds a few things. It copies behavior from something defined elsewere. This is a recipe for havoc. Here's a few links which discuss this in more detail:

https://softwareengineering.stackexchange.com/questions/134097/why-should-i-prefer-composition-over-inheritance https://en.wikipedia.org/wiki/Composition_over_inheritance https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

So if possible, give composition a try. Say you have `MyType`, and it has some function `f` defined on it. This means that you can extend `MyType` by making it a field in another type:

```
In [ ]: mutable struct MyType2
            mt::MyType
            ... # Other stuff
        end

        f(mt2::MyType2) = f(mt2.mt)
```

The pro here is that it's explicit: you've made the choice for each extension. The con is that this can require some extra code, though this can be automated by metaprogramming.

What if you really really really want inheritance of fields? There are solutions via metaprogramming. One simple solution is the `@def` macro.

```
In [25]: macro def(name, definition)
             return quote
                 macro $name()
                     esc($(Expr(:quote, definition)))
                 end
             end
         end

Out[25]: @def (macro with 1 method)
```

This macro is very simple. What it does is compile-time copy/paste. For example:

```
In [26]: @def give_it_a_name begin
             a = 2
             println(a)
         end
```

```
Out[26]: @give_it_a_name (macro with 1 method)
```

defines a macro `@give_it_a_name` that will paste in those two lines of code wherever it is used. For example, the reused fields of Optim.jl's solvers could be put into an `@def`:

```
In [27]: @def add_generic_fields begin
                 method_string::String
                 n::Int64
                 x::Array{T}
                 f_x::T
                 f_calls::Int64
                 g_calls::Int64
                 h_calls::Int64
         end

Out[27]: @add_generic_fields (macro with 1 method)
```

and those fields can be copied around with

```
In [28]: mutable struct LBFGSState{T}
             @add_generic_fields
             x_previous::Array{T}
             g::Array{T}
             g_previous::Array{T}
             rho::Array{T}
             # ... more fields ...
         end
```

Because `@def` works at compile-time, there is no cost associated with this. Similar metaprogramming can be used to build an "inheritance feature" for Julia. One package which does this is ConcreteAbstractions.jl which allows you to add fields to abstract types and make the child types inherit the fields:

```
In [ ]: # The abstract type
        @base mutable struct AbstractFoo{T}
            a
            b::Int
            c::T
            d::Vector{T}
        end

        # Inheritance
        @extend mutable struct Foo <: AbstractFoo
            e::T
        end
```

where the `@extend` macro generates the type-definition:

```
In [ ]: mutable struct Foo{T} <: AbstractFoo
            a
            b::Int
            c::T
            d::Vector{T}
            e::T
        end
```

But it's just a package? Well, that's the beauty of Julia. Most of Julia is written in Julia, and Julia code is first class and performant (here, this is all at compile-time, so again runtime is not affected at all). Honestly, if something ever gets added to Julia's Base library for this, it will likely look very similar, and the only real difference to the user will be that the compiler will directly recognize the keywords, meaning you would use base and extend instead of `@base` and `@extend`. So if you have something that really really really needs inheritance, go for it: there's no downsides to using a package + macro for this. But you should really try other means to reduce the runtime information and build a more performant and more Julian architecture first.

1.7 Conclusion

Programming for type systems has a different architecture than object-oriented systems. Instead of being oriented around the objects and their fields, type-dispatch systems are oriented around the actions of types. Using traits, multiple inheritance behavior can be given. Using this structure, the compiler can have maximal information, and use this to optimize the code. But also, this directly generalizes the vast majority of the code to not be "implementation-dependent", allowing for duck-typed code to be fully performance, with all of the details handled by dispatch/traits/abstract types. The end result is flexible, generic, and high performance code.