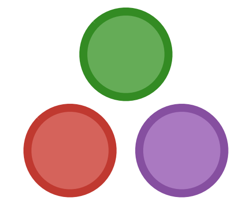
The Unreasonable Effectiveness of Multiple Dispatch

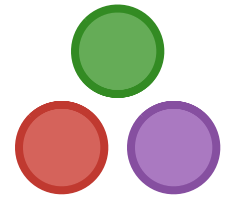
Stefan Karpinski

Multiple Dispatch?



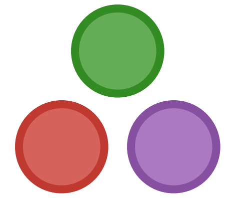
dispatch degree	syntax	dispatched on arguments	selection power
none	f(x, y)	{ }	$O(1)$
single	x.f(y)	{x}	$O(\|X\|)$
multiple	f(x, y)	{x, y}	$O(\|X\| \cdot \|Y\|)$

Multiple Dispatch?



A demo is the most effective explanation

Unreasonable Effectiveness?



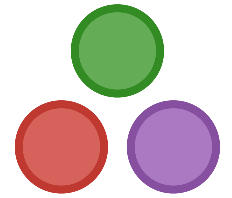
If you're familiar with Julia's ecosystem, you may have noticed…

- there's a really large amount of **code sharing** and **code reuse**

    as compared to comparable high-level dynamic languages

    (already a pretty happy-to-share crowd)

Delightful & Puzzling



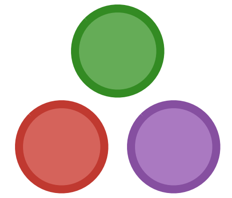
What is going on? Why is there such an increase in code reuse?

- it's a **genuine surprise**—we did not predict most this
- we believe this is **due to multiple dispatch**
- we chose multiple dispatch to have this effect, chose it because…
    1. it's very **natural for mathematics**

       meaning of x + y depends on x and y not just x
    2. it's great for expressing **generic algorithms**

       this is actually part of the explanation but not all of it

Two Kinds of Code Reuse



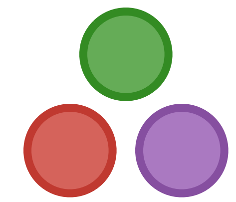
There are two quite different kinds of code reuse that we see

1. **common types** shared by very different packages
2. **generic algorithms** applied to many different types

These are different and have **different explanations**

- both stem from aspects of multiple dispatch

Sharing Common Types



Example shared type problem:

- suppose you have an **RGB type** much like the one in ColorTypes.jl

  it's simple: it bundles a red, a green and a blue value together

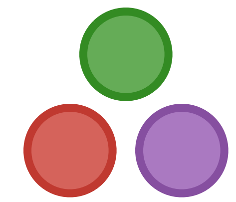
  for simplicity, let's say it's non-parametric—r/g/b fields are Float64

- it comes with some **basic operations** that make sense to the author

Suppose additionally that someone else wants to **add operations**

- this is a pretty simple and reasonable thing to want to do
- example: https://github.com/JuliaGraphics/ColorVectorSpace.jl

Sharing Common Types



In Julia, how does this work?

- just **add methods** to RGB in your own code

  that's it, there's no problem

  example: ColorVectorSpaces

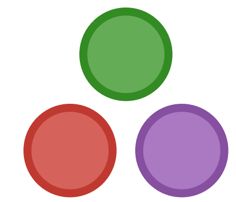
- works for **existing operations**

```
Base.zero(::Type{RGB}) = RGB(0,0,0)
```

- works for **new operations**

```
# coefficients from squaring conversion to grayscale and normalizing
dotc(x::RGB, y::RGB) = 0.200*x.r*y.r + 0.771*x.g*y.g + 0.029*x.b*y.b
```

Sharing Common Types



What's the big deal? Is this really harder in other languages?

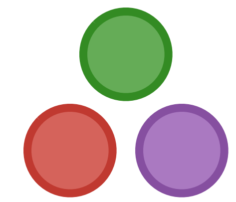
- surprisingly, yes—especially in **class-based object-oriented** ones

    we'll call these languages **"CBOO"** for short

In a CBOO languages, methods go "inside" of classes

- methods are literally defined **textually inside** of the class definition
- to add methods to a class, you have two choices:
    1. **edit the original class** and add methods there
    2. **inherit from the original class** and add methods there

Sharing Common Types



Adding every method to the RGB class is problematic

- you have to **convince the author** that it's a good idea

  they may be reluctant since they'll have to maintain your code

- if everyone convinces them, the **class become huge**

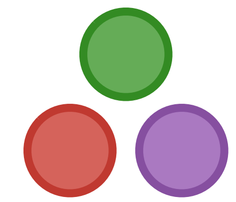
  you're probably not the only one who wants to add some stuff

- **you can't change your mind** without potentially breaking every user

  e.g. ColorVectorSpaces appears to be an abandoned experiment

  in Julia, anyone who doesn't load ColorVectorSpaces is unaffected

Sharing Common Types



Inheriting from the RGB class is just as problematic

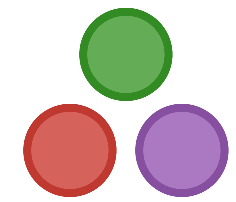
- it needs a **new name**—say MyRGB—instead of just RGB
- my operations won't apply to plain RGB objects **created by others**

  there are techniques to deal with this with fancy names like "Dependency Injection" and "Inversion of Control" but they are a **pain in the butt**

- using multiple extensions together requires **multiple inheritance**

  if there's MyRGB and YourRGB need OurRGB that **inherits from both** in order to use them together—assuming the language can even do that

Sharing Common Types

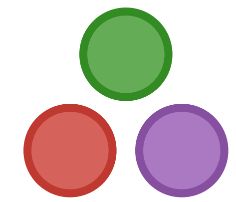


So in CBOO we have to choose between two lousy options

- there are actually **two more options** but they are also bad

1. **give up on dispatch**
    - use external functions: `f(x, y)` instead of `x.f(y)`
    - f can be defined outside of class in separate code base
    - gives up all code selection power also (ruins other kind of reuse)
2. **give up on code sharing**
    - just make your own version of RGB
    - can call it whatever you want, including RGB
    - often the best option in CBOO languages 😢

Sharing Common Types



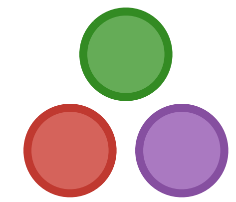
The key capability in Julia that allows sharing common types is:

- **you can define methods on types after the type is defined**
- can be done in a separate package which can be loaded or not

Additional subtleties:

- generic functions are properly namespaces unlike methods in CBOO
- i.e. `MyPackage.foo` and `YourPackage.foo` are separate functions

Generic Code

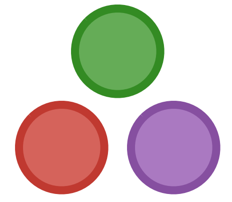


Example generic algorithm:

```
using LinearAlgebra

function f(A, vs)
    t = zero(eltype(A))
    for v in vs
        t += inner(v, A, v)  # <= multiple dispatch
    end
end

inner(v, A, v) = dot(v, A*v)  # very generic defintion
```

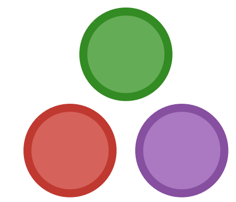
Pro tip: to write highly generic code, just leave off all types!

Generic Code



Let’s play with the code to understand it

Generic Code



Let's go a step further

- let's define a **new type** to which this code applies
- we'll define a **one-hot vector** type

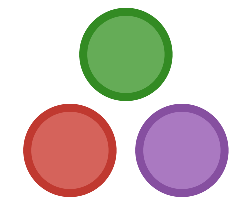
  represents a vector with a single 1 and otherwise 0 entries

  $v = \langle 0, \ldots, 0, 1, 0, \ldots, 0 \rangle$

  commonly used in machine learning

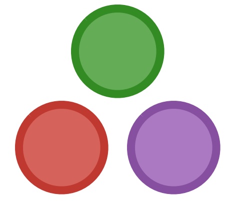
  can be represented **very compactly**

Generic Code: OneHotVector type



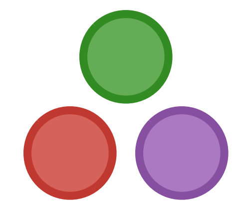
```
import Base: size, getindex, *

struct OneHotVector <: AbstractVector{Bool}
    len :: Int
    ind :: Int
end

# define some methods

size(v::OneHotVector) = (v.len,)

getindex(v::OneHotVector, i::Integer) = i == v.ind
```

Generic Code: OneHotVector



Back to the playground… er, REPL

Generic Code: inner analysis



Let's zoom in on `inner(v, A, v)`:

```
inner(v, A, v) = dot(v, A*v)
```

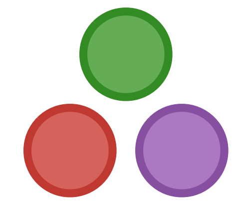
Breaking down the computation:

- `A*v` calls a generic matrix multiplication implementation
  - iterates through columns of A and multiplies them by each entry in v
  - returns a **copy of column of A** with type `Vector{Float64}`
- `dot(v, A*v)` calls a generic dot implementation

  does indexing into `v::OneHotVector` and `A*v::Vector{Float64}`

We can do much better based on our knowledge of `OneHotVector`!

Generic Code: optimizing matvec



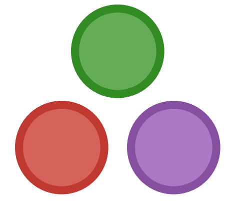
For `OneHotVectors` all `A*v` is doing is selecting a column

- optimizing this in Julia is extremely simple
- just define the right method for the `*` function

This new method definition is all that's required:

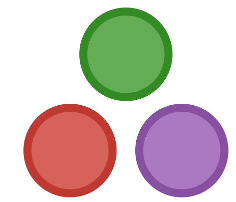
```
*(A::AbstractMatrix, v::OneHotVector) = A[:, v.ind]
```

Generic Code: optimizing matvec



Let's take a look at matvec optimized

Generic Code: optimizing `inner`



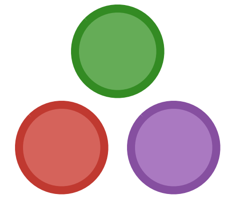
But we can do even better for `inner(v, A, w)`

- for `OneHotVectors` just does scalar indexing into `A`
- just define a method for the right combination of arguments

This new method definition is all that's required:

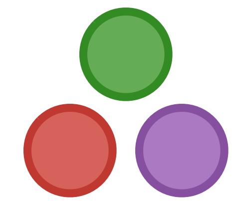
```
inner(v::OneHotVector, A, w::OneHotVector) = A[v.ind, w.ind]
```

Generic Code: optimizing inner



Let's take a look at inner optimized

Generic Code: not just for optimization



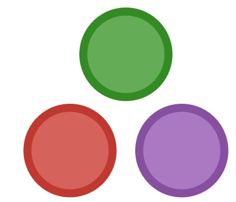
In these cases multiple dispatch was used for speed:

- were slower-than-optimal but correct fallbacks
- generic `*` provided by Julia
- generic `inner` provided by us — `dot(v, A*w)`

Sometimes there is no generic implementation

- you will get a method error
- use multiple dispatch to provide missing functionality

Generic Code: single dispatch comparison



It's possible but there are a lot of problems…

- `*(A::AbstractMatrix, v::OneHotVector) = A[:, v.ind]`

Problem: need to **dispatch on 2nd argument** not the 1st

- `AbstractMatrix.*` can do **"double dispatch"**

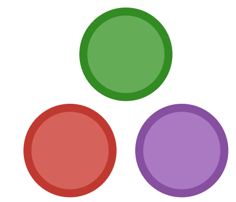
  `AbstractMatrix.*` calls `v.__rmul__(A)` or (or something like that)

- in Python this pattern is standard and the name is `v.__rmul__`

  this is what default `*` does in Python already — but only for `+` and `*`

- in C++ and other languages you have to roll your own

Generic Code: single dispatch comparison



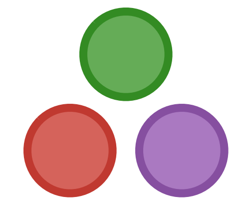
It's possible but there are a lot of problems…

- `inner(v::OneHotVector, A, w::OneHotVector) = A[v.ind, w.ind]`

Problem: need to **dispatch on 1st and 3rd arguments**

- unclear how to do this in a single dispatch language
- "triple dispatch"? not a thing anyone actually does
- no real solution in single-dispatch languages

Generic Code: method overloading



What about method overloading in C++/Java/C# etc.?

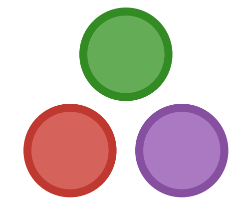
- can write `inner(v::OneHotVector, A, w::OneHotVector)`

  doesn't that solve the problem?

No: the **method doesn't get called** when the caller is generic

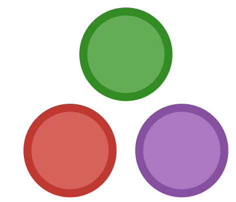
- generic means `v` and `w` have **abstract static type** like `AbstractVector`
- above method is only called for **concrete static type** `OneHotVector`

Multiple Dispatch!



dispatch degree	syntax	dispatched on arguments	selection power
none	f(x, y)	$\{\}$	$O(1)$
single	x.f(y)	$\{x\}$	$O(\|X\|)$
multiple	f(x, y)	$\{x, y\}$	$O(\|X\| \cdot \|Y\|)$

Generic Code: single dispatch comparison



How real is the problem?

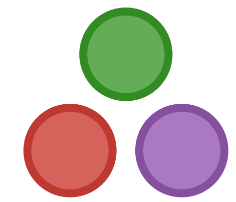
- generic code like this **occurs in the wild** in Julia all the time
- anecdotally, this kind of generic code **“just works”**

  the biggest problem is usually people “overtyping” their code
- this is **the main difference** from other languages

Therefore:

- it **does seem to matter** and **multiple dispatch** is the solution

Conclusion



Unusually large amounts of **code reuse and sharing** in Julia

Two varieties, both explained by aspects of **multiple dispatch**:

1. **common types** shared by very different packages

   Reason: **methods can be added to types after they are defined**

2. **generic algorithms** applied to many different types

   Reason: **methods are selected based on all argument types**