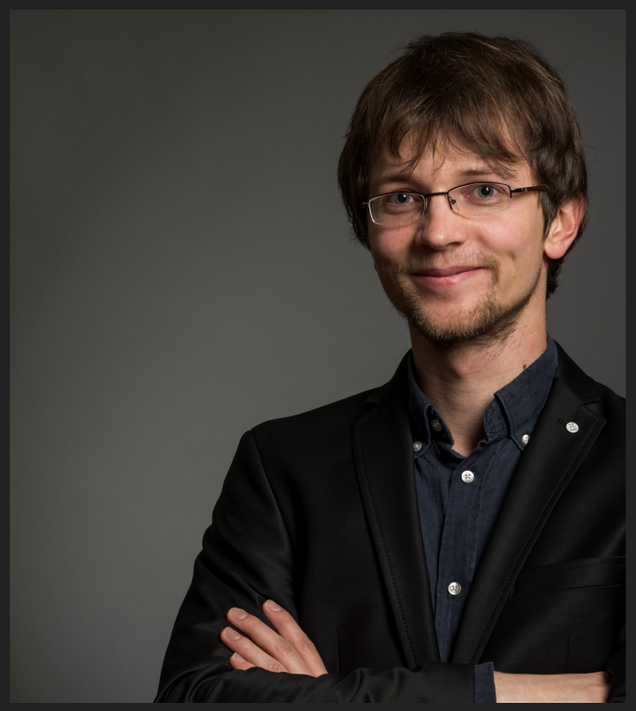
A NEW TRAITS.JL – WHERETRAITS.JL

Easily dispatch on whatever you want



WHO I AM:
- STEPHAN SAHM
- DATASCIENCE CONSULTANT IN MUNICH
- MACHINE LEARNING REPLY

WHAT I LIKE TO DO:
- APPLIED ML, PROBABILISTIC STUFF
- ABSTRACT PROGRAMMING THEORY (FUNCTIONAL AND CATEGORY THEORY)

ABSTRACT

With traits you can extend your function dispatch from plain type-hierarchies to in principle whatever you want.

A typical example is to dispatch on whether an iterable knows its length or not, and create respective specialized code. You cannot do this via standard type-hierarchies, as anything can be an iterable by just implementing iterate.

WhereTraits.jl gives you an intuitive and extendable way to define such dispatch by making use of julia's where syntax.

BIASED COMPARISON OF TRAITS SOLUTIONS IN JULIA

- Traits abstraction are always zero-cost

Plain Julia (aka Holy Traits)

```julia
function collect_iterable(iterable)
    _collect_iterable(Base.IteratorSize(iterable), iterable)
end

function _collect_iterable(::Union{Base.HasLength, Base.HasShape}, iterable)
    # check if empty
    # preallocate array
    # fill array
end
```

- Best Error Handling
- Unlimited features
- no fixed contracts
- Traits dispatch only extendable by PackageOwner
- Tedious to extend

SimpleTraits.jl

```julia
using SimpleTraits
@traitdef IsNice{X}
@traitdef BelongTogether{X,Y} # traits can have several parameters

# Explicitly add types to a trait-group
@traitimpl IsNice{Int}
@traitimpl BelongTogether{Int,String}
# Use function instead
# @traitimpl IsNice{X} <- isnice(X)
# isnice(X) = false # set default

# check trait
using Test
@test istrait(IsNice{Int})
@test !istrait(BelongTogether{Int,Int})

# dispatch on trait
@traitfn f(x::X) where {X; IsNice{X}} = "Very nice!"
@traitfn f(x::X) where {X; !IsNice{X}} = "Not so nice!"
@test f(5)=="Very nice!"
@test f(5.)=="Not so nice!"
```

- Arbitrary arity
- Syntax to define traits
- Syntax to dispatch on traits
- no fixed contracts
- Cannot overload same function twice with different traits
- Traits dispatch only extendable by PackageOwner
- Tedious to extend

BinaryTraits.jl

```julia
# Use package and import desired positive/negative trait type aliases
using BinaryTraits
using BinaryTraits.Prefix: Can

# Define a trait and its interface contracts
@trait Fly
@implement Can{Fly} by fly(_, destination::Location, speed::Float64)

# Define your data type and implementation
struct Bird end
fly(::Bird, destination::Location, speed::Float64) = "Wohoo! Arrived! 🎯"

# Assign your data type to a trait
@assign Bird with Can{Fly}

# Verify that your implementation is correct
@check(Bird)

# Dispatch for all flying things
@traitfn flap(::Can{Fly}, freq::Float64) = "Flapping wings at $freq Hz"
```

- Syntax to ensure traits contract
- Composite Traits which also inherit the contract
- Syntax to define traits
- Syntax to dispatch on traits
- Cannot overload same function twice with different traits
- Only binary traits
- Traits dispatch only extendable by PackageOwner

CanonicalTraits.jl

```julia
"""vector space to scalar space"""
function V2F end

@trait VecSpace{F, V} where
  {F = V2F(V)} begin
   vec_add    :: [V, V] => V
   scalar_mul :: [F, V] => V
end

@trait VecSpace{F, V} >: InnerProd{F, V} where
  {F = V2F(V)} begin
  dot :: [V, V] => F
end

@trait InnerProd{F, V} >: Ortho{F, V} where
  {F = V2F(V)} begin
  gram_schmidt! :: [V, Vector{V}] => V

  gram_schmidt!(v :: V, vs :: Vector{V}) where V = begin
    for other in vs
        coef = dot(v, other) / dot(other, other)
        incr = scalar_mul(-coef, other)
        v = vec_add(v, incr)
    end
    magnitude = sqrt(dot(v, v))
    scalar_mul(1/magnitude, v)
  end
end
```

- Arbitrary arity
- Syntax to ensure traits contract
- Composite Traits which also inherit the contract
- Nice Error Handling
- Syntax to define traits
- Syntax to dispatch on traits
- quite complex syntax
- Cannot overload same function twice with different traits (function is bound to trait)
- Traits dispatch only extendable by PackageOwner

WhereTraits.jl (my Traits.jl renamed)

```julia
using WhereTraits
# dispatch on functions returning Bool
@traits f(a) where {isodd(a)} = (a+1)/2
@traits f(a) where {!isodd(a)} = a/2
f(4) # 2.0
f(5) # 3.0
# dispatch on functions returning anything
@traits g(a) where {Base.IteratorSize(a)::Base.HasShape} = 43
@traits g(a) = 1
g([1,2,3]) # 43
g(Iterators.repeated(1)) # 1
# dispatch on bounds on functions returning Types
@traits h(a) where {eltype(a) <: Number} = true
@traits h(a) = false
h([1.0]) # true
h([""]) # false
```

- Arbitrary arity
- Unlimited features
- Functions can be extended for multiple different traits, EVEN FROM OTHER PACKAGES
- easy to extend (using exactly same syntax)
- support for doc
- Syntax to dispatch on traits (one-macro-only)
- Error messages are currently still difficult-to-read MethodErrors

WHERETRAITS.JL DETAILS

github.com/schlichtanders/WhereTraits.jl

FEATURES

- extendable (also from other packages)
- use julia functions as traits
- Intuitive simple syntax, combining outer and inner function into one
- Dispatch on arbitrary many functions (be careful about method-ambiguity-errors, same as when using standard holy-traits-plain-julia)
- Doc support
- Precompilation

IMPLEMENTATION

- Just macros
- Global information needed, stored in functions themselves
- Each standard-dispatch defines another outer-function, leading to less conflicts between definitions

LIMITATIONS

- Many warnings due to needed safe redefinition and no way to supress them (anyone any idea?)
- Error Messages are plain MethodErrors (looking for how to improve this)

Plain Julia (aka Holy Traits)

```
function collect_iterable(iterable)
    _collect_iterable(Base.IteratorSize(iterable), iterable)
end

function _collect_iterable(::Union{Base.HasLength, Base.HasShape}, iterable)
    # check if empty
    # preallocate array
    # fill array
end
```

WhereTraits.jl

```
using WhereTraits
const MyHasLength = Union{Base.HasLength, Base.HasShape}
@traits function collect_iterable(iterable) where {Base.IteratorSize(iterable)::MyHasLength}
    # check if empty
    # preallocate array
    # fill array
end
```