Package 'lambda.r'

May 18, 2018

Type Package

Title Modeling Data with Functional Programming

Version 1.2.3

Date 2018-05-17

Depends R (>= 3.0.0)

Imports formatR

Suggests testit

Author Brian Lee Yung Rowe

Maintainer Brian Lee Yung Rowe <r@zatonovo.com>

Description A language extension to efficiently write functional programs in R. Syntax extensions include multi-part function definitions, pattern matching, guard statements, built-in (optional) type safety.

License LGPL-3

LazyLoad yes

NeedsCompilation no

Repository CRAN

Date/Publication 2018-05-17 23:32:53 UTC

R topics documented:

`lambda.r-package` *Modeling Data with Functional Programming*

Description

Lambda.r is a language extension that supports a functional programming style in R. As an alternative to the object-oriented systems, lambda.r offers a functional syntax for defining types and functions. Functions can be defined with multiple distinct function clauses similar to how multipart mathematical functions are defined. There is also support for pattern matching and guard expressions to finely control function dispatching, all the while still supporting standard features of R. Lambda.r also introduces its own type system with intuitive type constructors are and type constraints that can optionally be added to function definitions. Attributes are also given the attention they deserve with a clean and convenient syntax that reduces type clutter.

Details

Package:	lambda.r
Type:	Package
Version:	1.2.3
Date:	2018-05-17
License:	LGPL-3
LazyLoad:	yes

Data analysis relies so much on mathematical operations, transformations, and computations that a functional approach is better suited for these types of applications. The reason is that object models rarely make sense in data analysis since so many transformations are applied to data sets. Trying to define classes and attach methods to them results in a futile enterprise rife with arbitrary choices and hierarchies. Functional programming avoids this unnecessary quandry by making objects and functions first class and preserving them as two distinct entities.

R provides many functional programming concepts mostly inherited from Scheme. Concepts like first class functions and lazy evaluation are key components to a functional language, yet R lacks some of the more advanced features of modern functional programming languages. Lambda.r introduces a syntax for writing applications using a declarative notation that facilitates reasoning about your program in addition to making programs modular and easier to maintain.

Function Definition: Functions are defined using the `%as%` (or `%:=%`) symbol in place of `<-`. Simple functions can be defined as simply

```
f(x) %as% x
```

and can be called like any other function.

```
f(1)
```

Functions that have a more complicated body require braces.

```
f(x) %as% { 2 * x }
```

```
g(x, y) %:=% {
  z <- x + y
  sqrt(z)
}
```

Infix notation: Functions can be defined using infix notation as well. For the function g above, it can be defined as an infix operator using
x %g% y %:=% z <- x + y sqrt(z)

Multipart functions and guards: Many functions are defined in multiple parts. For example absolute value is typically defined in two parts: one covering negative numbers and one covering everything else. Using guard expressions and the `%when%` keyword, these parts can be easily captured.

```
abs(x) %when% { x < 0 } %as% -x
abs(x) %as% x
```

Any number of guard expressions can be in a guard block, such that all guard expressions must evaluate to true.

```
abs(x) %when% {
  is.numeric(x)
  length(x) == 1
  x < 0
} %as% -x

abs(x) %when% {
  is.numeric(x)
  length(x) == 1
} %as% x
```

If a guard is not satisfied, then the next clause is tried. If no function clauses are satisfied, then an error is thrown.

Pattern matching: Simple scalar values can be specified in a function definition in place of a variable name. These scalar values become patterns that must be matched exactly in order for the function clause to execute. This syntactic technique is known as pattern matching.
Recursive functions can be defined simply using pattern matching. For example the famed Fibonacci sequence can be defined recursively.

```
fib(0) %as% 1
fib(1) %as% 1
fib(n) %as% { fib(n-1) + fib(n-2) }
```

This is also useful for conditionally executing a function. The reason you would do this is that it becomes easy to symbolically transform the code, making it easier to reason about.

```
pad(x, length, TRUE) %as% c(rep(NA,length), x)
pad(x, length, FALSE) %as% x
```

It is also possible to match on `NULL` and `NA`.

```
sizeof(NULL) %as% 0
sizeof(x) %as% length(x)
```

Types: A type is a custom data structure with meaning. Formally a type is defined by its type constructor, which codifies how to create objects of the given type. The lambda.r type system is fully compatible with the built-in S3 system. Types in lambda.r must start with a capital letter.

Type constructors: A type constructor is responsible for creating objects of a given type. This is simply a function that has the name of the type. So to create a type `Point` create its type constructor.

```
Point(x,y) %as% list(x=x,y=y)
```

Note that any built-in data structure can be used as a base type. Lambda.r simply extends the base type with additional type information.
Types are then created by calling their type constructor.

```
p <- Point(3,4)
```

To check whether an object is of a given type, use the `%isa%` operator.

```
p %isa% Point
```

Type constraints: Once a type is defined, it can be used to limit execution of a function. R is a dynamically typed language, but with type constraints it is possible to add static typing to certain functions. S4 does the same thing, albeit in a more complicated manner.
Suppose we want to define a distance function for `Point`. Since it is only meaningful for `Points` we do not want to execute it for other types. This is achieved by using a type constraint, which declares the function argument types as well as the type of the return value. Type constraints are defined by declaring the function signature followed by type arguments.

```
distance(a,b) %::% Point : Point : numeric
distance(a,b) %as% { sqrt((b$x - a$x)^2 + (b$y - a$y)^2) }
```

With this type constraint `distance` will only be called if both arguments are of type `Point`. After the function is applied, a further requirement is that the return value must be of type `numeric`. Otherwise lambda.r will throw an error. Note that it is perfectly legal to mix and match lambda.r types with S3 types in type constraints.

Type variables: Declaring types explicitly gives a lot of control, but it also limits the natural polymorphic properties of R functions. Sometimes all that is needed is to define the relationship between arguments. These relationships can be captured by a type variable, which is simply any single lower case letter in a type constraint.
In the distance example, suppose we do not want to restrict the function to just `Points`, but whatever type is used must be consistent for both arguments. In this case a type variable is sufficient.

```
distance(a,b) %::% z : z : numeric
distance(a,b) %as% { sqrt((b$x - a$x)^2 + (b$y - a$y)^2) }
```

The letter `z` was used to avoid confusion with the names of the arguments, although it would have been just as valid to use `a`.
Type constraints and type variables can be applied to any lambda.r function, including type constructors.

The ellipsis type: The ellipsis can be inserted in a type constraint. This has interesting properties as the ellipsis represents a set of arguments. To specify that input values should be captured by the ellipsis, use `...` within the type constraint. For example, suppose you want a function that multiplies the sum of a set of numbers. The ellipsis type tells lambda.r to bind the types associated with the ellipsis type.

```
sumprod(x, ..., na.rm=TRUE) %::% numeric : ... : logical : numeric
sumprod(x, ..., na.rm=TRUE) %as% { x * sum(..., na.rm=na.rm) }

> sumprod(4, 1,2,3,4)
[1] 40
```

Alternatively, suppose you want all the values bound to the ellipsis to be of a certain type. Then you can append "'..." to a concrete type.

```
sumprod(x, ..., na.rm=TRUE) %::% numeric : numeric... : logical : numeric
sumprod(x, ..., na.rm=TRUE) %as% { x * sum(..., na.rm=na.rm) }

> sumprod(4, 1,2,3,4)
[1] 40
> sumprod(4, 1,2,3,4,'a')
Error in UseFunction(sumprod, "sumprod", ...) :
  No valid function for 'sumprod(4,1,2,3,4,a)'
```

If you want to preserve polymorphism but still constrain values bound to the ellipsis to a single type, you can use a type variable. Note that the same rules for type variables apply. Hence a type variable represents a type that is not specified elsewhere.

```
sumprod(x, ..., na.rm=TRUE) %::% a : a... : logical : a
sumprod(x, ..., na.rm=TRUE) %as% { x * sum(..., na.rm=na.rm) }

> sumprod(4, 1,2,3,4)
[1] 40
> sumprod(4, 1,2,3,4,'a')
Error in UseFunction(sumprod, "sumprod", ...) :
  No valid function for 'sumprod(4,1,2,3,4,a)'
```

The don't-care type: Sometimes it is useful to ignore a specific type in a constraint. Since we are not inferring all types in a program, this is an acceptable action. Using the "'." within a type constraint tells lambda.r to not check the type for the given argument.
For example in `f(x, y) %::% . : numeric : numeric`, the type of `x` will not be checked.

Attributes: The attribute system in R is a vital, yet often overlooked feature. This orthogonal data structure is essentially a list attached to any object. The benefit of using attributes is that it reduces the need for types since it is often simpler to reuse existing data structures rather than create new types.

Suppose there are two kinds of `Points`: those defined as Cartesian coordinates and those as Polar coordinates. Rather than create a type hierarchy, you can attach an attribute to the object. This keeps the data clean and separate from meta-data that only exists to describe the data.

```
Point(r,theta, 'polar') %as% {
  o <- list(r=r,theta=theta)
  o@system <- 'polar'
  o
}

Point(x,y, 'cartesian') %as% {
  o <- list(x=x,y=y)
  o@system <- 'cartesian'
  o
}
```

Then the `distance` function can be defined according to the coordinate system.

```
distance(a,b) %::% z : z : numeric
```

```
distance(a,b) %when% {
  a@system == 'cartesian'
  b@system == 'cartesian'
} %as% {
  sqrt((b$x - a$x)^2 + (b$y - a$y)^2)
}

distance(a,b) %when% {
  a@system == 'polar'
  b@system == 'polar'
} %as% {
  sqrt(a$r^2 + b$r^2 - 2 * a$r * b$r * cos(a$theta - b$theta))
}
```

Note that the type constraint applies to both function clauses.

Debugging: As much as we would like, our code is not perfect. To help troubleshoot any problems that exist, lambda.r provides hooks into the standard debugging system. Use `debug.lr` as a drop-in replacement for `debug` and `undebug.lr` for `undebug`. In addition to being aware of multipart functions, lambda.r's debugging system keeps track of what is being debugged, so you can quickly determine which functions are being debugged. To see which functions are currently marked for debugging, call `which.debug`. Note that if you use `debug.lr` for all debugging then lambda.r will keep track of all debugging in your R session. Here is a short example demonstrating this.

```
> f(x) %as% x
> debug.lr(f)
> debug.lr(mean)
>
> which.debug()
[1] "f"    "mean"
```

Note

Stable releases are uploaded to CRAN about once a year. The most recent package is always available on github [2] and can be installed via 'rpackage' in 'crant' [3].

```
rpackage https://github.com/zatonovo/lambda.r/archive/master.zip
```

Author(s)

Brian Lee Yung Rowe

Maintainer: Brian Lee Yung Rowe <r@zatonovo.com>

References

[1] Blog posts on lambda.r: http://cartesianfaith.com/category/r/lambda-r/

[2] Lambda.r source code, https://github.com/muxspace/lambda.r

[3] Crant, https://github.com/muxspace/crant

See Also

%as%, describe, debug.lr, %isa%

Examples

```
is.wholenumber <-
  function(x, tol = .Machine$double.eps^0.5)  abs(x - round(x)) < tol

## Use built in types for type checking
fib(n) %::% numeric : numeric
fib(0) %as% 1
fib(1) %as% 1
fib(n) %when% {
  is.wholenumber(n)
} %as% {
  fib(n-1) + fib(n-2)
}

fib(5)


## Using custom types
Integer(x) %when% { is.wholenumber(x) } %as% x

fib.a(n) %::% Integer : Integer
fib.a(0) %as% Integer(1)
fib.a(1) %as% Integer(1)
fib.a(n) %as% { Integer(fib.a(n-1) + fib.a(n-2)) }

fib.a(Integer(5))


## Newton-Raphson optimization
converged <- function(x1, x0, tolerance=1e-6) abs(x1 - x0) < tolerance
minimize <- function(x0, algo, max.steps=100)
{
  step <- 0
  old.x <- x0
  while (step < max.steps)
  {
    new.x <- iterate(old.x, algo)
    if (converged(new.x, old.x)) break
    old.x <- new.x
  }
  new.x
}

iterate(x, algo) %::% numeric : NewtonRaphson : numeric
iterate(x, algo) %as% { x - algo$f1(x) / algo$f2(x) }

iterate(x, algo) %::% numeric : GradientDescent : numeric
iterate(x, algo) %as% { x - algo$step * algo$f1(x) }
```

```
NewtonRaphson(f1, f2) %as% list(f1=f1, f2=f2)
GradientDescent(f1, step=0.01) %as% list(f1=f1, step=step)


fx <- function(x) x^2 - 4
f1 <- function(x) 2*x
f2 <- function(x) 2

algo <- NewtonRaphson(f1,f2)
minimize(3, algo)

algo <- GradientDescent(f1, step=0.1)
minimize(3, algo)
```

`duck-typing` *Functions for duck typing*

Description

Duck typing is a way to emulate type checking by virtue of an object's characteristics as opposed to strong typing.

Usage

```
argument %isa% type
argument %hasa% property
argument %hasall% property
```

Arguments

`argument`	An object to inspect
`type`	A type name
`property`	A property of an object

Details

These operators provide a convenient method for testing for specific properties of an object.

`%isa%` checks if an object is of the given type.

`%hasa%` checks if an object has a given property. This can be any named element of a list or data.frame.

Value

Boolean value indicating whether the specific test is true or not.

Author(s)

Brian Lee Yung Rowe

See Also

%as%

Examples

```
5 %isa% numeric

Point(r,theta, 'polar') %as% {
  o <- list(r=r,theta=theta)
  o@system <- 'polar'
  o
}

p <- Point(5, pi/2, 'polar')
p
```

---

introspection *Introspection for lambda.r*

Description

These tools are used for debugging and provide a means of examining the evaluation order of the function definitions as well as provide a lambda.r compatible debugger.

Usage

```
debug.lr(x)

undebug.lr(x)

is.debug(fn.name)

which.debug()

undebug.all()

describe(...)
## S3 method for class 'lambdar.fun'
print(x, ...)
## S3 method for class 'lambdar.type'
print(x, ...)
```

Arguments

x	The function
fn.name	The name of the function
...	Additional arguments

Details

For a basic description of the function it is easiest to just type the function name in the shell. This will call the print methods and print a clean output of the function definition. The definition is organized based on each function clause. If a type constraint exists, this precedes the clause signature including guards. To reduce clutter, the actual body of the function clause is not printed. To view a clause body, each clause is prefixed with an index number, which can be used in the `describe` function to get a full listing of the function.

```
describe(fn, idx)
```

The 'debug.lr' and 'undebug.lr' functions are replacements for the built-in debug and undebug functions. They provide a mechanism to debug a complete function, which is compatible with the dispatching in lambda.r. The semantics are identical to the built-ins. Note that these functions will properly handle non-lambda.r functions so only one set of commands need to be issued.

Lambda.r keeps track of all functions that are being debugged. To see if a function is currently set for debugging, use the `is.debug` function. To see all functions that are being debugged, use `which.debug`. It is possible to undebug all debugged functions by calling `undebug.all`.

Value

The defined functions are invisibly returned.

Author(s)

Brian Lee Yung Rowe

Examples

```
## Not run:
f(x)
debug.lr(f)
which.debug()
undebug.lr(f)

## End(Not run)
```

`UseFunction` *Primary dispatcher for functional programming*

Description

UseFunction manages the dispatching for multipart functions in lambda.r. This is used internally by lambda.r.

Usage

```
UseFunction(fn, fn.name, ...)
NewObject(type.fn, type.name, ...)
```

Arguments

`fn`	The function reference that is being applied
`fn.name`	The name of a function that uses functional dispatching. This is just the name of the function being defined
`type.fn`	The function representing the type constructor
`type.name`	The name of the type
`...`	The arguments that are passed to dispatched functions

Details

This function is used internally and generally does not need to be called by an end user.

Value

Returns the value of the dispatched function

Author(s)

Brian Lee Yung Rowe

See Also

`%as%`

Examples

```
# Note that these are trivial examples for pedagogical purposes. Due to their
# trivial nature, most of these examples can be implemented more concisely
# using built-in R features.


reciprocal(x) %::% numeric : numeric
reciprocal(x) %when% {
```

```
  x != 0
} %as% {
  1 / x
}

reciprocal(x) %::% character : numeric
reciprocal(x) %as% {
  reciprocal(as.numeric(x))
}

seal(reciprocal)

print(reciprocal)
reciprocal(4)
reciprocal("4")
```

`%as%` *Define functions and type constructors in lambda.r*

Description

The %as% function is used in place of the assignment operator for defining functions and type constructors with lambda.r. The %as% operator is the gateway to a full suite of advanced functional programming features.

Usage

```
signature %::% types
signature %as% body
seal(fn)
```

Arguments

`signature`	The function signature for the function to be defined
`types`	The type constraints for the function
`body`	The body of the function
`fn`	The function to seal

Details

The %as% and %::% operators are the primary touch points with lambda.r.

Functions are defined using %as% notation. Any block of code can be in the function definition. For simple criteria, pattern matching of literals can be used directly in lambda.r. Executing different function clauses within a multipart function sometimes requires more detail than simple pattern matching. For these scenarios a guard statement is used to define the condition for execution. Guards are simply an additional clause in the function definition defined by the %when% operator.

```
fib(n) %when% { n >= 0 } %as% { fib(n-1) + fib(n-2) }
```

A function variant only executes if the guard statements all evaluate to true. As many guard statements as desired can be added in the block. Just separate them with either a new line or a semi-colon.

Type constructors are no different from regular functions with one exception: the function name must start with a capital letter. In lambda.r, types are defined in PascalCase and functions are lower case. Violating this rule will result in undefined behavior. The return value of the type constructor is the object that represents the type. It will have the type attached to the object.

```
Number(x, set='real') %as% {     x@set <- set     x   }
```

Attributes can be accessed using lambda.r's at-notation, which borrows from S4's member notation. These attributes are standard R attributes and should not be confused with object properties. Hence with lambda.r it is possible to use both the $ to access named elements of lists and data.frames while using the @ symbol to access the object's attributes.

Type constraints specify the type of each input argument in addition to the return type. Using this approach ensures that the arguments can only have compatible types when the function is called. The final type in the constraint is the return type, which is checked after a function is called. If the result does not have the correct return type, then the call will fail. Each type is separated by a colon and their order is defined by the order of the function clause signature.

Each function clause can have its own type constraint. Once a constraint is defined, it will continue to be valid until another type constraint is defined.

'seal' finalizes a function definition. Any new statements found will reset the definition, effectively deleting it. This is useful to prevent other people from accidentally modifying your function definition.

Value

The defined functions are invisibly returned.

Author(s)

Brian Lee Yung Rowe

Examples

```
# Type constraints are optional and include the return type as the
# final type
reciprocal(x) %::% numeric : numeric
reciprocal(0) %as% stop("Division by 0 not allowed")

# The type constraint is still valid for this function clause
reciprocal(x) %when% {
  # Guard statements can be added in succession
  x != 0
  # Attributes can be accessed using '@' notation
  is.null(x@dummy.attribute)
} %as% {
  # This is the body of the function clause
  1 / x
}
```

```
# This new type constraint applies from this point on
reciprocal(x) %::% character : numeric
reciprocal(x) %as% {
  reciprocal(as.numeric(x))
}

# Seal the function so no new definitions are allowed
seal(reciprocal)

print(reciprocal)
reciprocal(4)
reciprocal("4")
```

Index