Package 'futile.paradigm'

February 19, 2015

Type Package

Title A framework for working in a functional programming paradigm in R

Version 2.0.4

Date 2012-02-06

Depends futile.options, RUnit

Author Brian Lee Yung Rowe

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

Description Provides dispatching implementations suitable for functional programming paradigms. The framework provides a mechanism for attaching guards to functions similar to Erlang, while also providing the safety of assertions reminiscent of Eiffel.

License LGPL-3

LazyLoad yes

Repository CRAN

Date/Publication 2012-02-06 16:53:54

NeedsCompilation no

R topics documented:

`futile.paradigm-package` *A framework for working in a functional programming paradigm in R*

Description

Provides dispatching implementations suitable for functional programming paradigms. The framework provides a mechanism for attaching guards to functions similar to Erlang, while also providing the safety of assertions reminiscent of Eiffel.

Details

Package:	futile.paradigm
Type:	Package
Version:	2.0.4
Date:	2012-02-06
License:	LGPL-3
LazyLoad:	yes

In many ways R is a functional language. The futile.paradigm complements R's functional strengths to make it easier to develop software in this paradigm. This package implements a function dispatching method based on the guard concept (in lieu of direct pattern matching) for functional programming in R. In contrast to UseMethod, which is used for S3 object-oriented method dispatching, futile.paradigm introduces UseFunction for functional programming. Since UseMethod only detects the type of the first argument, dispatching can be tricky/cumbersome when multiple arguments and/or types are supported. Also, the direct manipulation of class attributes is dangerous in the S3 paradigm. In the futile.paradigm, access is restricted via encapsulation to reduce the fragility of such a mechanism. In lieu of the S4 style that requires significant up-front design, the futile.paradigm intends to ease the development process from initial experimentation to systems development. This package provides a richer syntax for defining functions including the constraints that must be satisfied in order to dispatch to the particular function. A happy consequence of this approach is that code becomes self-documenting and functions are more concise since there is a clean separation of concerns between the true logic of a function and the data management tasks within the function.

Concrete implementations of the abstract function, must be defined in a hierarchical naming scheme based on the name of the abstract function. Similar to UseMethod, the concrete function name is derived from the abstract function with a unique suffix separated by a dot. Unlike UseMethod, the suffix is arbitrary as the dispatching and association is controlled by the guard and the number of arguments in the concrete function. Guards are expressions using the same variables as the concrete function variant or explicit functions operating on the same number of arguments as the concrete function it is guarding. Either form of a guard must return a boolean value. Multiple guard functions can be defined in the guard call and in this situation all guards must resolve to TRUE for the function to execute. In addition to a function, a default function can be defined by setting the guard expression to TRUE.

To use the package, two key operators are introduced to implement guard statements and assertions. Guards are declared by using the '%when%' operator, which is reminiscent of the Erlang approach to guards. By convention the guard is declared before the function definition as this communicates immediately the criteria required to execute this particular implementation of the parent function. See the examples section for a trivial implementation.

In addition to guard statements, this package provides post-assertions reminiscent of Eiffel's design-by-contract approach. Using the '%must%' operator validates function outputs and fails if they aren't satisfied. The arguments to a generated assertion are ordered with the result object first (named 'result'), then all of the function arguments.

Conventions: . Types are named in PascalCase . Functions are named in lower case with words separated by underscore . Function namespaces are separated by dot . Declare guard statements before concrete implementations . Declare ensure statements after concrete implementations . Avoid explicit UseFunction definitions . Avoid default function implementations unless a package defines all possible variants

Note

Due to scoping rules with operators, logical expressions need to be wrapped in parentheses.

Author(s)

Brian Lee Yung Rowe

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

References

Some background on using guards and pattern matching:

http://en.wikipedia.org/wiki/Guard_%28computing%29

See Also

UseFunction, %when%, %must%

Examples

```
# The guard must be defined before the concrete function variant
abs_max %when% is.numeric(a)
abs_max %also% (length(a) > 1)
# This adds a post-assertion to ensure the value is what you expect
abs_max %must% (result > 0)
abs_max %as% function(a) abs_max(a[1], abs_max(a[2:length(a)]))

abs_max %when% (is.numeric(a) && length(a) == 1)
abs_max %must% (result == a)
abs_max %as% function(a) a

abs_max %when% (a %isa% DataObject & a %hasa% data)
abs_max %as% function(a) abs_max(as.numeric(strsplit(a$data, ',')[[1]]))

abs_max %when% (is.numeric(a) & is.numeric(b))
```

```
abs_max %must% (result >= a | result >= b)
abs_max %as% function(a, b) max(abs(a), abs(b))

# Using a guard of TRUE acts as a default condition
abs_max %when% TRUE
abs_max %as% function(a,b) abs_max(as.numeric(a), as.numeric(b))

# Define constructor for DataObject
create.DataObject <- function(T, data, name=NA) list(name=name, data=data)

# Test some output
abs_max(2,-3) # Calls abs_max.twoArg

abs_max("3","-4") # Calls abs_max.twoArgDefault

abs_max(3,"-4") # Calls abs_max.twoArgDefault

a <- rnorm(10)
abs_max(a) # Calls abs_max.recursive1

b <- create(DataObject, c('12,-3,-5,8,-13,3,1,3'))
abs_max(b) # Calls abs_max.csv


## 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 %when% (algo %isa% NewtonRaphson)
iterate %as% function(x, algo) x - algo$f1(x) / algo$f2(x)

iterate %when% (algo %isa% GradientDescent)
iterate %as% function(x, algo) x - algo$step * algo$f1(x)

create.GradientDescent <- function(T, f1, step=0.01) list(f1=f1,step=step)


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

algo <- create(NewtonRaphson, f1=f1,f2=f2)
```

```
minimize(3, algo)

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

`AbuseMethod` *Dispatcher for high-level API functions*

Description

Used for library authors defining very high-level API functions. Not typically needed for normal development.

Usage

```
AbuseMethod(fn.name, type, ..., EXPLICIT = FALSE, ALWAYS = TRUE)
```

Arguments

`fn.name`	The parent function name. This is just the name of the original function
`type`	The target type or a regular value from which the type is derived
`...`	Arguments to pass to the dispatched function
`EXPLICIT`	Whether the type is passed in explicitly or not
`ALWAYS`	Whether a default function should be called if all else fails

Details

This alternative dispatching is for specialized purposes. It allows certain syntactic sugar not possible in UseMethod when performing method dispatching. If none of the above made sense, then don't use this function. Otherwise, it can be useful when defining very high-level functions that define interfaces for an API.

In the future this function may take on additional functionality to manage dispatching certain functions based on computer/network architecture.

Value

Returns the result of the dispatched function. When ALWAYS is FALSE, no default function will be called. Instead, an error message is printed. In certain circumstances this fail-fast behavior is preferred over the default lenient behavior.

Author(s)

Brian Lee Yung Rowe

See Also

`UseMethod`, `UseFunction`

Examples

```
# Trivial example for pedagogical reasons only
product <- function(...) AbuseMethod('product', ...)

product.numeric <- function(a,b) a * b
product.matrix <- function(a,b) a

product(4,2)
```

---

crud *API for CRUD-like operations*

Description

Provides a high-level API for abstracting CRUD operations for arbitrary objects. Currently only create is provided, as this is essential for type management within futile.paradigm. Others will be added as necessary.

Usage

```
create(type, ...)
create.default(type, ...)
```

Arguments

type	The object type to create as a symbol or character
...	Additional arguments to pass to dispatched functions

Details

Adding to 'create' requires a minimal function definition as the harness is provided in the package. Typically a list is returned and the function defines any defaults needed. This is similar to the S4 style but is simpler and not a requirement for using the rest of futile.paradigm.

In general the futile.paradigm avoids strings where syntax is explicit enough that this is possible. By convention types are PascalCased, which makes identifying a type even clearer. For this to work properly, it is essential that class names are not defined as types as defined symbols are assumed to be valid for dereferencing.

Value

'create' returns an object of the requested type.

Author(s)

Brian Lee Yung Rowe

Examples

```
create.Car <- function(T, wheels=4, doors=4) list(wheels=wheels, doors=doors)

# This is how you inherit from a type
create.SportsCar <- function(T, doors=2, ...)
  create(Car, doors=doors, ...)

my.car <- create(Car, doors=5)

your.car <- create('SportsCar')

# The type can be passed in via a variable as well
her.class <- 'Car'
her.car <- create(her.class, wheels=3)

my.car %isa% Car
your.car %isa% SportsCar
```

`UseFunction` *Primary dispatcher for functional programming*

Description

UseFunction is an alternative approach to function dispatching vis-a-vis UseMethod. It is designed for people interested in writing functional programs in R as opposed to object-oriented programs.

Usage

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

Arguments

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

Details

In most situations (i.e. following the conventions outlined by the futile.paradigm), explicit use of UseFunction is unnecessary as the system will automatically generate and execute this command as necessary. The only time it is necessary is when an abstract function has an ambiguous name (typically containing extraneous dots).

Explicit function definitions follow a simple template: fn.var <- function(...) UseFunction('fn.var', ...)

When calling the function, if no guards match, then an error is returned.

Value

Returns the value of the dispatched function

Note

For high-level API development, AbuseMethod may be more appropriate

Author(s)

Brian Lee Yung Rowe

See Also

AbuseMethod

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.

# Optional
#reciprocal <- function(...) UseFunction('reciprocal', ...)

reciprocal %when% is.numeric(x)
reciprocal %also% (x != 0)
reciprocal %must% (sign(result) == sign(x))
reciprocal %as% function(x) 1 / x

reciprocal %when% is.character(x)
reciprocal %as% function(x) reciprocal(as.numeric(x))

print(reciprocal)
reciprocal(4)
```

`%must%` *Add post-assertion validations to a function to define the conditions when results of a child function are valid*

Description

The 'ensure' function defines the conditions for successful execution for the given function. This is an optional declaration that requires a previous guard declaration for the given function.

The 'ensures' function provides reflection and displays assertions defined for a function hierarchy.

Usage

```
fn.ref %must% condition
ensures(fn, inherits = TRUE, child = NULL)
```

Arguments

fn.ref	This is the function for which the assertion is applied. Unlike with 'guard', the function must exist
condition	The conditions for dispatching to this function. This can either be an expression, a function, or vector of functions. See Details for more information
fn	The function to find assertions for. If this is a child function, the parent function will be queried
inherits	If a function is passed that has no assertions, whether to search for a parent function. Typically this is safe to leave as the default
child	Used to filter the assertions for a specific concrete function. Typically safe to ignore

Details

Combining guards with post-execution assertions provides a framework for design-by-contract programming. This paradigm forces developers to define the programming interface for each function explicitly in the code. Doing so ensures that failures are detected immediately (known as fail-fast) reducing troubleshooting time. Using the 'ensure' command is optional.

When using an expression in the ensure command, the executed function is called with the following arguments: the result of the concrete function (named 'result') followed by the arguments passed to the concrete function. If the condition fails, execution will halt with an error message.

Value

The '%must%' declaration is used for its side-effects and has no return value.

The 'ensures' function works like 'guards' and displays all registered assertions for a given abstract function.

Author(s)

Brian Lee Yung Rowe

See Also

UseFunction, %when%

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.
```

```
# The expression must operate on arguments declared in the concrete function.
logarithm %when% is.numeric(x)
logarithm %as% function(x) logarithm(x, exp(1))

# Explicit form (only necessary for special cases)
logarithm %when% is.numeric(x)
logarithm %also% is.numeric(y)
logarithm %must% (! is.nan(result) && ! is.infinite(result))
logarithm %as% function(x,y) log(x, base=y)

logarithm %when% TRUE
logarithm %as% function(x,y) logarithm(as.numeric(x), as.numeric(y))

logarithm(5)

# Uses all arguments in assertion
f %when% (is.numeric(a) & is.numeric(b) & b > 1)
f %must% (result == a + b)
f %as% function(a,b) a + b

f(2,3)

# View the function variants for this abstract function
ensures(logarithm)
```

`%when%` *Add guards to a function to define the conditions when a child function will execute*

Description

The '%when%' operator defines the conditions for execution for the given function. To use a function within futile.paradigm, a guard must be defined for each child function, even if it's a default guard using TRUE. Each '%when%' clause indicates a new function variant for a given function.

The '%also%' operator specifies additional clauses that must be met to execute the given function variant. It is provided as a syntactic convenience to shorten long guards.

'%as%' is used to assign the function variant definition to the function

'%default%' is used to specify a fallback if necessary

The '%isa%' operator performs type checking.

The '%hasa%' operator checks whether an object contains a named value.

The '%hasall%' function is like '%hasa%' but returns the conjunction of the '%hasa%' results.

The 'guards' function provides introspection and displays guards defined for a function hierarchy.

To access a specific function variant, use the 'variant' function. This function takes a name as an argument, returning a function reference. The primary purpose of this is for debugging.

Usage

```
fn.ref %when% condition
fn.ref %also% condition
fn.ref %as% fn.def

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

guards(fn.ref, inherits = TRUE)
variant(name.fn)
```

Arguments

fn.ref	The function for which a guard is applied. It does not need to exist yet
fn.def	An actual function definition to associate with the function variant
condition	The conditions for dispatching to this function. This can either be an expression, a function, or vector of functions. See Details for more information
inherits	If a function is passed that has no guards, whether to search for a parent function. Typically this is safe to leave as the default
type	A symbol or character describing the type to match
argument	The argument to match the type or properties with
property	The property or properties to look for within the object
name.fn	A function name that references a specific function variant

Details

Guards provide a mechanism for dispatching function variants consistent with the declarative style of functional programming. Using guards for dispatching has many benefits: function variants are self-contained and focus on a single task (separation of concerns), design-by-contract is inherent, data manipulation is separate from computational logic, self-documenting.

The futile.paradigm makes adoption of guards simple. A function is decorated with declarative statements, and futile.paradigm performs the necessary wiring. Any number of function variants can be attached to a given function, with each variant starting with the '%when%' operator:

<function> %when% (<logical expression>)

The guard statement tells futile.paradigm under what conditions this particular function variant should be called. This is defined by passing an expression to the guard function that operates on the arguments of the concrete function (see Examples). This expression is dynamically bound to an actual function at run-time. Guards are naturally scoped based on the number of arguments in a function. Hence only functions with the same number of arguments as were passed into the parent function will be considered for dispatching.

Named arguments passed into the parent function must match the named arguments defined in the child function. Hence, dispatching is defined by both the number of arguments and the matching named arguments, which introduces a greater level of control in dispatching than otherwise possible.

In general, the built-in argument matching will Do the Right Thing w.r.t. unnamed arguments, although the first match will be the one applied, which could result in unexpected behavior if one is careless.

Each function variant has exactly one '%when%' clause. For convenience, additional conjunctive clauses (i.e. they all must resolve to true) can be attached to the variant with the '%also%' operator:

<function> %also% (<logical expression>)

The order that the function guards are defined determines the order that functions are evaluated for satisfaction of guard criteria. This behavior is typically called top-to-bottom, left-to-right evaluation. The left-to-right evaluation only applies to the explicit form when multiple function guards are defined in a vector. These details are important to understand as default functions defined too early will be greedy and no other criteria will be evaluated.

Binding an actual function definition to this particular guard sequence is done by using the %as% operator. Each variant needs at minimum a %when% statement and an %as% statement to successfully bind a new variant to the given function.

In addition to variants with guards, the %default% operator can be used to specify a default function that is used if no function variants match when calling the function.

Another important consideration is that using the ellipsis argument is not supported in futile.paradigm. This is by design as the functional programming approach intentionally makes function arguments explicit, such that the ellipsis argument should never be needed in a child function definition.

The ' whether 'argument' is a 'type'. This is more than syntactic sugar in guard sequences as it adds a level of protective indirection when accessing type information. Note that the preferred format is to supply a raw symbol to

the syntax without undue clutter.

The ' a particular named property. This operator also supports raw symbols as the argument.

Deprecated Functions The below functions are for version 1 of the futile.paradigm and are officially deprecated. They will be removed in future versions.

The 'isStrict' function is an introspective function that indicates whether the given child function has strict guards or not.

When strict guards are disabled, then only argument length will be applied as a precondition plus any conditions defined by the guards. In certain circumstances this may be desired, although in general strict guards provides greater control. Note that this feature is deprecated.

Value

No value is returned for '%when%' nor '%also%' since these operators are used purely for their side-effects.

The 'guards' function returns a list of guard functions for each child function defined. This essentially shows the evaluation path that UseFunction will take.

The 'variant' function returns a function reference if found.

'%isa%' returns a logical value indicating whether a variable is an instance of class.

'%hasa%' returns a logical value indicating whether a property exists in an object.

'%hasall%' returns the conjunction of all '%hasa%' results for an object. This is useful if a set of properties are required for execution.

Note

In general functions in FP do not have side-effects. This principle does not apply here since this function is used to implement the framework itself.

Author(s)

Brian Lee Yung Rowe

See Also

UseFunction, %must%

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.


# The expression must operate on arguments declared in the concrete function.
logarithm %when% is.numeric(x)
logarithm %as% function(x) logarithm(x, exp(1))

# Defaults are applied on a per-argument length basis
logarithm %when% TRUE
logarithm %as% function(x) logarithm(as.numeric(x))

logarithm %when% is.numeric(x)
logarithm %also% is.numeric(y)
logarithm %as% function(x,y) log(x, base=y)

logarithm %when% TRUE
logarithm %as% function(x,y) logarithm(as.numeric(x), as.numeric(y))

logarithm %default% function(...) cat("This is the default function\n")

logarithm(100,10)
logarithm(5)
logarithm("5")
logarithm(6,7,8)

# View the function variants for this abstract function
guards(logarithm)

# In the futile.paradigm, the convention is to name types in PascalCase
a <- create(Apple, seeds=103)
a
a
```

Index