Package 'pmatch'

December 21, 2018

Type Package

Title Pattern Matching

Version 0.1.4

Description Implements type constructions and pattern matching.
Using this package, you can specify a type of object and write functions
that matches against the structure of objects of such types to program
data structure transformations more succinctly.

License GPL-3

ByteCompile true

Encoding UTF-8

RoxygenNote 6.1.0

Language en-GB

Depends R (>= 3.2)

Imports dplyr, purrr, rlang (>= 0.2.0), tibble, glue, foolbox,
magrittr

Suggests shiny, DT, covr, styler, lintr, testthat, knitr, rmarkdown,
microbenchmark, ggraph, tidygraph, tailr

URL https://github.com/mailund/pmatch

BugReports https://github.com/mailund/pmatch/issues

VignetteBuilder knitr

NeedsCompilation no

Author Thomas Mailund [aut, cre]

Maintainer Thomas Mailund <mailund@birc.au.dk>

Repository CRAN

Date/Publication 2018-10-19 15:20:02 UTC

R topics documented:

.. 2
:= 3
assert_correctly_formed_pattern_expression 4
bind 4
cases 5

cases_expr_ 6
case_func 6
construction_printer 7
copy_env 8
deparse_construction 8
make_match_expr 9
process_alternatives 9
process_arg 10
process_arguments 10
process_constructor 11
process_constructor_constant 11
process_constructor_function 12
test_pattern_ 12
test_pattern_rec 13
transform_cases_call 14
transform_cases_function 14
[<-.pmatch_bind 15

Index **16**

`..` *Function for specifying a sequence of patterns/expressions*

Description

This function is used when you want to test for more than one pattern in parallel.

Usage

```
..(...)
```

Arguments

`...`	The patterns/expressions to combine

Details

If you want to test two or more patterns against the same number of values, then you can use this function to combine the values and expressions in a `cases()` (or related) function.

Examples

```
llist := NIL | CONS(car, cdr : llist)
compare_llists <- function(l1, l2) {
    cases(..(l1, l2),
          ..(NIL, NIL) -> TRUE,
          ..(NIL, .)   -> FALSE,
          ..(., NIL)   -> FALSE,
          ..(CONS(car1,cdr1), CONS(car2,cdr2)) ->
              car1 == car2 && compare_llists(cdr1, cdr2)
          )
}
compare_llists(NIL, NIL)
```

```
compare_llists(CONS(1, NIL), CONS(1, NIL))
compare_llists(CONS(1, NIL), NIL)
compare_llists(CONS(1, NIL), CONS(2, NIL))
```

`:=` *Define a new data type from a sequence of constructors.*

Description

This assignment operator introduces a domain-specific language for specifying new types. Types are defined by the ways they can be constructed. This is provided as a sequence of `|`-separated constructors, where a constructor is either a constant, i.e., a bare symbol, or a function.

Usage

```
":="(data_type, constructors)
```

Arguments

`data_type`	The name of the new data type. Should be given as a bare symbol.
`constructors`	A list of `|`-separated constructor specifications.

Details

We can construct an enumeration like this:

```
numbers := ONE | TWO | THREE
```

This will create the type `numbers` and three constants, `ONE`, `TWO`, and `THREE` that can be matched against using the `cases` function

```
x <- TWO cases(x, ONE -> 1, TWO -> 2, THREE -> 3)
```

Evaluating the `cases` function will compare the value in `x` against the three patterns and recognize that `x` holds the constant `TWO` and it will then return `2`.

With function constructors we can create more interesting data types. For example, we can create a linked list like this

```
linked_list := NIL | CONS(car, cdr : linked_list)
```

This expression defines constant `NIL` and function `CONS`. The function takes two arguments, `car` and `cdr`, and requires that `cdr` has type `linked_list`. We can create a list with three elements, 1, 2, and 3, by writing

```
CONS(1, CONS(2, CONS(3, NIL)))
```

and we can, e.g., test if a list is empty using

```
cases(lst, NIL -> TRUE, CONS(car,cdr) -> FALSE)
```

A special pattern, `otherwise`,can be used to capture all patterns, so the emptiness test can also be written

```
cases(lst, NIL -> TRUE, otherwise -> FALSE)
```

Arguments to a constructor function can be typed. To specify typed variables, we use the `:`-operator. The syntax is then `var : type`. The type will be checked when you construct a value using the constructor.

Examples

```
linked_list := NIL | CONS(car, cdr : linked_list)
lst <- CONS(1, CONS(2, CONS(3, NIL)))
len <- function(lst, acc = 0) {
    cases(lst,
          NIL -> acc,
          CONS(car,cdr) -> len(cdr, acc + 1))
}
len(lst)

list_sum <- function(lst, acc = 0) {
    cases(lst,
          NIL -> acc,
          CONS(car,cdr) -> list_sum(cdr, acc + car))
}
list_sum(lst)
```

---

assert_correctly_formed_pattern_expression

Raise an error if a match expression is malformed.

Description

Raise an error if a match expression is malformed.

Usage

```
assert_correctly_formed_pattern_expression(match_expr)
```

Arguments

match_expr	The match expression

---

bind

Dummy object used for generic function dispatching.

Description

Dummy object used for generic function dispatching.

Usage

```
bind
```

Format

An object of class `pmatch_bind` of length 1.

cases *Dispatches from an expression to a matching pattern*

Description

Given an expression of a type defined by the `:=` operator, `cases` matches it against patterns until it find one that has the same structure as `expr`. When it does, it evaluates the expression the pattern is associated with. During matching, any symbol that is not quasi-quoted will be considered a variable, and matching values will be bound to such variables and be available when an expression is evaluated.

Usage

```
cases(expr, ...)
```

Arguments

`expr`	The value the patterns will be matched against.
`...`	A list of `pattern -> expression` statements.

Value

The value of the expression associated with the first matching pattern.

See Also

`:=`

Examples

```
linked_list := NIL | CONS(car, cdr : linked_list)
lst <- CONS(1, CONS(2, CONS(3, NIL)))
len <- function(lst, acc = 0) {
    cases(lst,
          NIL -> acc,
          CONS(car,cdr) -> len(cdr, acc + 1))
}
len(lst)

list_sum <- function(lst, acc = 0) {
    cases(lst,
          NIL -> acc,
          CONS(car,cdr) -> list_sum(cdr, acc + car))
}
list_sum(lst)
```

cases_expr_ *Create an expression that tests patterns against an expression in turn*

Description

Where cases evaluates expressions based on pattern matches, this function creates a long if-else expression that tests patterns in turn and evaluate the expression for a matching pattern. This function is intended for meta-programming rather than usual pattern matching.

Usage

```
cases_expr_(expr, ...)

cases_expr(expr, ...)
```

Arguments

expr	The expression to test against. This is usually a bare symbol.
...	Pattern matching rules as in cases.

Functions

- cases_expr_: Version that expects expr to be quoted.
- cases_expr: Version that quotes expr itself.

Examples

```
linked_list := NIL | CONS(car, cdr : linked_list)

length_body <- cases_expr(
                  lst,
                  NIL -> acc,
                  CONS(car, cdr) -> ll_length(cdr, acc + 1))
length_body

ll_length <- rlang::new_function(alist(lst=, acc = 0), length_body)
ll_length(CONS(1, CONS(2, CONS(3, CONS(4, NIL)))))
```

case_func *Creates a pattern matching function.*

Description

Creates a function that can contain expressions of a type defined by the := operator. The first argument of the generated function will be matched against patterns provided in the ... parameter of this function.

Usage

```
case_func(...)
```

Arguments

...	A list of variables for the function in addition the data to be matched against which will automatically added plus `pattern -> expression` statements.

Details

When you call the generated function, and the first argument is matching a pattern, it evaluates the expression the pattern is associated with. During matching, any symbol that is not quasi-quoted will be considered a variable, and matching values will be bound to such variables and be available when an expression is evaluated.

Functions created with `case_func` do not support the `..` operator, but you can always create constructors for fixed-number tuples, e.g.

```
tuples := ..(first, second) | ...(first, second, third)
```

Be careful not to use `.` here, if you use dot as a generic variable.

Value

A function that can pattern match

See Also

`:=`

Examples

```
linked_list := NIL | CONS(car, cdr : linked_list)
lst <- CONS(1, CONS(2, CONS(3, NIL)))
len <- case_func(acc = 0,
   NIL -> acc,
   CONS(car,cdr) -> len(cdr, acc + 1)
)
len(lst)

list_sum <- case_func(acc = 0,
   NIL -> acc,
   CONS(car,cdr) -> list_sum(cdr, acc + car)
)
list_sum(lst)

tuples := ..(first, second) | ...(first, second, third)
f <- case_func(..(.,.) -> 2, ...(.,.,.) -> 3)
f(..(1, 2))
f(...(1, 2, 3))
```

`construction_printer` *Print a constructed value*

Description

Print a constructed value

Usage

```
construction_printer(x, ...)
```

Arguments

x	Object to print
...	Additional parameters; not used.

---

copy_env *Move the bound variables from one environment into another.*

Description

Move the bound variables from one environment into another.

Usage

```
copy_env(from, to, names = ls(from, all.names = TRUE))
```

Arguments

from	The environment we want to copy from.
to	The environment where we want to bind the variables.
names	Names of the variables to copy. By default, all of them.

---

deparse_construction *Create a string representation from a constructed object*

Description

Create a string representation from a constructed object

Usage

```
deparse_construction(x, ...)
```

Arguments

x	The object to translate into a string
...	Additional parameters; not used.

Value

A string representation of `object`

make_match_expr *Create an if-statement for* cases_expr *and* cases_expr_ *functions*

Description

Create an if-statement for `cases_expr` and `cases_expr_` functions

Usage

```
make_match_expr(expr, match_expr, continue)
```

Arguments

`expr`	The expression we pattern match against.
`match_expr`	The pattern specification, on the form pattern -> expression
`continue`	The expression that goes in the `else` part of the `if` expression. If this is `NULL`, we create an `if`-expression instead of an `if-else`-expression.

Value

A new if-expression

process_alternatives *Goes through a list of* |*-separated expressions and define them as constructors*

Description

Goes through a list of |-separated expressions and define them as constructors

Usage

```
process_alternatives(constructors, data_type_name, env)
```

Arguments

`constructors`	The constructs specification
`data_type_name`	The type the constructor should generate
`env`	The environment where we define the constructor

process_arg *Build a tibble form a list of constructor arguments.*

Description

Build a tibble form a list of constructor arguments.

Usage

```
process_arg(argument)
```

Arguments

argument	The argument provided to a constructor in its definition

Value

A tibble with a single row, the first column holds the argument name, the second its type.

process_arguments *Construct a tibble from all the arguments of a constructor*

Description

Construct a tibble from all the arguments of a constructor

Usage

```
process_arguments(constructor_arguments)
```

Arguments

constructor_arguments	The arguments provided in the constructor specification

Value

The arguments represented as a tibble. The first column contain argument names, the second their types.

process_constructor *Create a constructor and put it in an environment.*

Description

Create a constructor and put it in an environment.

Usage

```
process_constructor(constructor, data_type_name, env)
```

Arguments

constructor	The construct specification
data_type_name	The type the constructor should generate
env	The environment where we define the constructor

process_constructor_constant *Create a constant constructor and put it in an environment.*

Description

Create a constant constructor and put it in an environment.

Usage

```
process_constructor_constant(constructor, data_type_name, env)
```

Arguments

constructor	The construct specification
data_type_name	The type the constructor should generate
env	The environment where we define the constructor

process_constructor_function *Create a function constructor and put it in an environment.*

Description

Create a function constructor and put it in an environment.

Usage

```
process_constructor_function(constructor, data_type_name, env)
```

Arguments

constructor	The construct specification
data_type_name	The type the constructor should generate
env	The environment where we define the constructor

test_pattern_ *Test if a pattern matches an expression*

Description

Test if a value, expr, created from constructors matches a pattern of constructors. The test_pattern_ function requires that test_expr is a quoted expression while the test_pattern function expects a bare expression and will quote it itself.

Usage

```
test_pattern_(expr, test_expr, eval_env = rlang::caller_env(),
  match_parent_env = rlang::caller_env())

test_pattern(expr, test_expr, eval_env = rlang::caller_env(),
  match_parent_env = rlang::caller_env())
```

Arguments

expr	A value created using constructors.
test_expr	A constructor pattern to test expr against.
eval_env	The environment where constructors can be found.
match_parent_env	Environment to use as the parent of the match bindings we return. This parameter enables you provide additional values to the environment where match-expressions are evaluated.

Value

NULL if the pattern does not match or an environment with bound variables if it does.

Functions

- `test_pattern_`: Version that quotes `test_expr` itself.
- `test_pattern`: Version that quotes `test_expr` itself.

Examples

```
type := ZERO | ONE(x) | TWO(x,y)
zero <- ZERO
one <- ONE(1)
two <- TWO(1,2)

as.list(test_pattern(zero, ZERO)) # returns an empty binding
test_pattern_(one, quote(ZERO)) # returns NULL
as.list(test_pattern_(one, quote(ONE(v)))) # returns a binding for v
as.list(test_pattern(two, TWO(v,w))) # returns a binding for v and w
```

`test_pattern_rec` *Recursive comparison of expression and pattern.*

Description

Recursive comparison of expression and pattern.

Usage

```
test_pattern_rec(escape, expr, test_expr, eval_env, match_env)
```

Arguments

`escape`	Continuation from callCC, used to escape if we cannot match.
`expr`	The expression to match again.
`test_expr`	The pattern we are trying to match.
`eval_env`	The environment where we get constructors from.
`match_env`	The environment to put matched variables in.

Value

An environment containing bound variables from the expression, if matching. If the pattern doesn't match, the function escapes through the `escape` continuation.

transform_cases_call *Recursive function for transforming a call* cases.

Description

Recursive function for transforming a call cases.

Usage

```
transform_cases_call(expr, ...)
```

Arguments

expr	The expression to transform.
...	Additional callback arguments to make this work with foolbox

Value

Updated expression.

transform_cases_function *Transform a function containing a* cases *call into one that instead has if-statements.*

Description

Transform a function containing a cases call into one that instead has if-statements.

Usage

```
transform_cases_function(fun)
```

Arguments

fun	A function

Value

Another function with a transformed body

See Also

cases

`[<-.pmatch_bind` *Bind variables to pattern-matched expressions.*

Description

The `bind` object itself doesn't do anything. It simply exists in order to define notation for binding variables using the sub-script operator.

Usage

```
## S3 replacement method for class 'pmatch_bind'
dummy[...] <- value
```

Arguments

`dummy`	The `bind` object. Only used to dispatch to the right subscript operator.
`...`	Patterns to assign to.
`value`	Actual values to assign

Examples

```
bind[x, y] <- c(2,4)
x == 2
y == 4

llist := NIL | CONS(car, cdr : llist)
L <- CONS(1, CONS(2, CONS(3, NIL)))
bind[CONS(first, CONS(second, rest))] <- L
first == 1
second == 2
```

Index

∗Topic **datasets**
bind, 4
..., 2
:=, 3, *5–7*
[<-.pmatch_bind, 15

assert_correctly_formed_pattern_expression, 4

bind, 4

case_func, 6
cases, *3*, 5, *6*
cases(), *2*
cases_expr, *9*
cases_expr *(*cases_expr_*)*, 6
cases_expr_, 6, *9*
construction_printer, 7
copy_env, 8

deparse_construction, 8

make_match_expr, 9

process_alternatives, 9
process_arg, 10
process_arguments, 10
process_constructor, 11
process_constructor_constant, 11
process_constructor_function, 12

test_pattern *(*test_pattern_*)*, 12
test_pattern_, 12
test_pattern_rec, 13
transform_cases_call, 14
transform_cases_function, 14