`fp` — library for functional programming

Table of contents

The functions of the `fp` package are higher order functions and other utilities useful for functional programming. Some other functions useful for functional programming are already contained in the MuPAD standard library, like `map`, `select` and `zip`.

For a more detailed description of concepts like "higher order function", "currying" and "fixed points" see for example the textbook "Computability, Complexity and Languages" by M. Davis, R. Sigal, and E. J. Weyuker, Academic Press (1994).

Most of the functions of the `fp` package take functions as arguments and return other functions. In this context a function may be a functional environment, a procedure, a kernel function or any other object which may be regarded as a function (i.e. applied to arguments). Note that almost all MuPAD objects are functions in this sense.

The rational integer `2/3` for example may be regarded as a constant function returning the value `2/3`:

```
>> 2/3(x)
```

$$2/3$$

The list `[sin, cos, 2/3]` may be regarded as a unary function mapping `x` to `[sin(x), cos(x), 2/3]`:

```
>> [sin, cos, 2/3](x)
```

$$[\sin(x), \cos(x), 2/3]$$

The call of the package functions occurs e.g. via `fp::fixedpt(f)`. Due to this mechanism naming conflicts with other library functions are avoided. If this is found to be too awkward the methods of the `fp` package may be exported. After calling `export(fp, fixedpt)` the function `fixedpt` is also directly available, i.e. `fixedpt(f)` may also be called. If a variable with the name `fixedpt` already exists then `export` raises an error. The value of the identifier `fixedpt` must then be deleted in order to be exported. With `export(fp)` all methods of the `fp` package are exported.

`fp::apply` – apply function to arguments

`fp::apply(f,a)` returns `f(a)`.

Call(s):

- `fp::apply(f <, e, ...>)`

Parameters:

`f` — function
`e` — object used as argument

Return Value: The result of the function call `f(e,...)`.

Side Effects: Same side effects as when calling `f(e,...)` directly.

Details:

- `fp::apply` applies the function `f` to the arguments given by `e, ...`.

Example 1. Apply the function `f` to `x` and `y`:

```
>> fp::apply(f, x, y)
                                f(x, y)
```

Example 2. Apply the functions of the first list to the arguments given by the second list:

```
>> zip([sin, cos], [x, y], fp::apply)
                            [sin(x), cos(y)]
```

Changes:

- No changes.

`fp::bottom` – the function that never returns

`fp::bottom()` never returns because it raises an error.

Call(s):

- `fp::bottom()`

Return Value: This function never returns.

Side Effects: Raises an error in any case.

Details:

- `fp::bottom` never returns because it raises an error.

Example 1.

```
>> fp::bottom()
 Error: bottom reached [fp::bottom]
```

Changes:

- No changes.

`fp::curry` – curry a n-ary function

`fp::curry(f)` returns the higher-order function $x \mapsto (y \mapsto f(x, y))$.

Call(s):

- `fp::curry(f <, n>)`

Parameters:

`f` — n-ary function
`n` — nonnegative integer

Return Value: A unary higher-order function.

Details:

- `fp::curry` returns the curried version of the `n`-ary function `f`. If no arity `n` is given, then the function is assumend to be binary.
- If `n` is smaller than 2 then `f` is returned. Otherwise, given a n-ary function f, `fp::curry` returns the function

$$x_1 \mapsto (x_2 \mapsto \ldots (x_n \mapsto f(x_1, \ldots, x_n)) \ldots)$$

Example 1. Create curried versions of binary and 3-nary functions:

```
>> cf := fp::curry(f):
   cf(x)(y)
```

```
f(x, y)
```

```
>> cg := fp::curry(g, 3):
   cg(x)(y)(z)
```

```
g(x, y, z)
```

Example 2. A curried version of `_plus` may be used to create a function which increments its argument by `1`:

```
>> inc := fp::curry(_plus)(1):
   inc(x)
```

```
x + 1
```

Changes:

- No changes.

`fp::expr_unapply` – create a functional expression from an expression

`fp::expr_unapply(e,x)` tries to interpret the expression `e` as a function in `x` and to return a functional expression computing that function.

Call(s):

- `fp::expr_unapply(e <, x, ...>)`

Parameters:

`e` — expression
`x` — identifier

Return Value: A functional expression or `FAIL`.

Related Functions: `fp::unapply`

Details:

- `fp::expr_unapply` views the expression `e` as a function in the indeterminates `x,...` and tries to return a functional expression computing that function. If `fp::expr_unapply` can not find a functional expression `FAIL` is returned.
- If no indeterminates are given, any indeterminates of `e` found by `indets` are used.

Example 1. Get the functional expression computing `sin(x)`:

```
>> fp::expr_unapply(sin(x), x)
                                  sin
```

Example 2. Get the functional expression computing `sin(x)^2+cos(x)^2`:

```
>> fp::expr_unapply(sin(x)^2 + cos(x)^2)
                               2      2
                            cos  + sin
```

Changes:

- `fp::expr_unapply` used to be `unapply`.

`fp::fixargs` – create function by fixing all but one argument

`fp::fixargs(f,1,y)` returns the function $x \mapsto f(x, y)$.

Call(s):

- `fp::fixargs(f, n <, e, ...>)`

Parameters:

`f` — function
`n` — positive integer defining free argument
`e` — object used as fixed argument

Return Value: An unary function.

Details:

- `fp::fixargs` returns an unary function, defined by fixing all but the `n`-th argument of the function `f` to the values given by `e...`.
- Thus, given a m-ary function f and $m-1$ values $e_1, \ldots, e_{m-1}$, `fp::fixargs` returns the function

$$x \mapsto f(e_1, \ldots, e_{n-1}, x, e_n, \ldots, e_{m-1})$$

Example 1. Fix the first and third argument of `f` to `x1` and `x3`:

```
>> fp::fixargs(f, 2, x1, x3)(y)

                          f(x1, y, x3)
```

Example 2. Create a function which increments its argument by one:

```
>> inc := fp::fixargs(_plus, 1, 1):
   inc(x)

                              x + 1
```

Example 3. Create a function which tests the identifier `x` for a type:

```
>> type_of_x := fp::fixargs(testtype, 2, x):
   map([DOM_INT, DOM_IDENT], type_of_x)

                          [FALSE, TRUE]
```

Changes:

- No changes.

`fp::fixedpt` – returns fixed point of a function

`fp::fixedpt(f)` returns the fixed point of the unary function `f`.

Call(s):

- `fp::fixedpt(f)`

Parameters:

`f` — unary function

Return Value: A unary function.

Details:

- `fp::fixedpt` returns the fixed point of the unary function `f`.
- `fp::fixedpt` is implemented as the Y combinator which is defined as follows:

$$Y : f \mapsto g(f)(g(f))$$

where the function g is defined as

$$g : f \mapsto h \mapsto x \mapsto f(h(h))(x)$$

Example 1. A function computing the Fibonacci numbers is created as a fixed point:

```
>> fb2 := (f,n) -> if n <= 2 then 1 else f(n-1) + f(n-2) end:
   fib := fp::fixedpt(fp::curry(fb2)):
   fib(i) $ i=1..9
```

```
                1, 1, 2, 3, 5, 8, 13, 21, 34
```

Changes:

- No changes.

`fp::fold` – create function which iterates over sequences

`fp::fold(f,e)` returns a function which repeatedly applies `f` to sequences of arguments, using `e` as starting value.

Call(s):

- `fp::fold(f <, e, ...>)`

Parameters:

`f` — function
`e` — object used as starting value

Return Value: A function.

Details:

- `fp::fold` returns a function which repeatedly applies `f` to sequences of arguments, where the expressions `e...` are used as starting values.
- Thus, given the function f and the starting values $e_1, \ldots, e_n$, `fp::fold` returns the function which is defined by

$$(x_1, x_2, \ldots, x_m) \mapsto f(x_m, \ldots, f(x_2, f(x_1, e1, \ldots, e_n)) \ldots)$$

  for any positive integer m. If the argument sequnce is void (i.e. $m = 0$) the function simply returns the sequence $(e_1, \ldots, e_n)$.

Example 1.

```
>> fp::fold(f, x)(y1, y2, y3)

                    f(y3, f(y2, f(y1, x)))
```

Example 2. The function `pset` returns the power set of the set given by its arguments:

```
>> addelem := (x,y) -> y union map(y, _union, {x}):
   pset := fp::fold(addelem, {{}}):
   pset(a,b,c)

     {{}, {b}, {a}, {c}, {b, c}, {a, b}, {a, c}, {a, b, c}}
```

Changes:

- No changes.

`fp::nest` – repeated composition of function

`fp::nest(f,n)` returns the `n`-fold repeated composition of the function `f`.

Call(s):

- `fp::nest(f, n)`

Parameters:

`f` — function
`n` — nonnegative integer

Return Value: A function.

Related Functions: `_fconcat`, `_fnest`, `fp::nestvals`

Details:

- `fp::nest(f,n)` returns the `n`-fold repeated composition of the function `f`.
- Thus, given the function `f`, `fp::nest` returns the identity function `id` if `n` is 0 and otherwise the function

$$f \circ f \circ \ldots \circ f$$

`n`-fold repeated.

- Note that `fp::nest` is obselete, one should use the `@@` operator or its functional form `_fnest` instead. It is only supported for compatibility with former versions of MuPAD.

Example 1. Apply the 3-fold repeated composition of `f` to `x`:

```
>> fp::nest(f, 3)(x)
```

```
f(f(f(x)))
```

Example 2. Numerically finding a fixed point of the function `cos` by repeated application:

```
>> p :=fp::nest(cos, 100)(1.0):
   p, cos(p)
```

```
0.7390851332, 0.7390851332
```

Changes:

- No changes.

`fp::nestvals` – repeated composition returning intermediate values

`fp::nestvals(f,n)` returns a function which applies the function `f` `n`-fold repeatedly to ist argument and returns the intermediate results.

Call(s):

- `fp::nestvals(f, n)`

Parameters:

`f` — function
`n` — nonnegative integer

Return Value: A function.

Related Functions: `_fconcat`, `_fnest`

Details:

- `fp::nestvals` returns a function which applies the function `f` 0- to `n`-fold repeated to its arguments and returns these $n+1$ values as a list.
- Thus `fp::nestvals` returns the function

$$x \mapsto [x, f(x), f(f(x)), \ldots f(f(\ldots f(x)\ldots))]$$

- The function returned is equivalent to `[ _fnest(f,i) $ i=0..n ]`, but more efficient.

Example 1. Apply `f` 3 times nested to `x`:

```
>> fp::nestvals(f, 3)(x)

                [x, f(x), f(f(x)), f(f(f(x)))]
```

Example 2. Apply `cos` 4 times nested to `1.0` and return the result and intermediate values:

```
>> fp::nestvals(cos, 4)(1.0)

  [1.0, 0.5403023059, 0.8575532159, 0.6542897905, 0.7934803588]
```

Changes:

- No changes.

`fp::unapply` – create a procedure from an expression

`fp::unapply(e,x)` interprets the expression `e` as a function in `x` and returns a procedure computing that function.

Call(s):

- `fp::unapply(e <, x, ...>)`

Parameters:

`e` — expression
`x` — identifier

Return Value: A procedure.

Overloadable by: `e`

Related Functions: `fp::expr_unapply`

Details:

- `fp::unapply` views the expression `e` as a function in the indeterminates `x,...` and returns a procedure computing that function.
- If no indeterminates are given, any indeterminates of `e` found by `indets` are used.

Example 1. Get the procedure computing `sin(x)^2+cos(y)^2`:

```
>> s := fp::unapply(sin(x)^2 + cos(y)^2, x, y)

                  (x, y) -> cos(y)^2 + sin(x)^2
```

Changes:

- `fp::unapply` is a new function.