Package 'foolbox'

December 15, 2018

Title Function Manipulation Toolbox

Version 0.1.1

Description Provides functionality for manipulating functions and translating them in metaprogramming.

Depends R (>= 3.2)

License GPL-3

Language en-GB

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

Imports magrittr, rlang (>= 0.3.0)

Suggests covr, testthat, knitr, rmarkdown, microbenchmark

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

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

VignetteBuilder knitr

NeedsCompilation no

Author Thomas Mailund [aut, cre]

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

Repository CRAN

Date/Publication 2018-12-15 22:20:18 UTC

R topics documented:

`<.foolbox_pipe` *This operator is used together with* `rewrites` *to transform a function after it is defined and before it is assigned to a name.*

Description

This operator is used together with `rewrites` to transform a function after it is defined and before it is assigned to a name.

Usage

```
## S3 method for class 'foolbox_pipe'
pipe < fn
```

Arguments

`pipe`	A specificiation of a a pipeline of transformations provided using the subscript operator to `rewrites()`.
`fn`	The function we wish to transform.

See Also

`[.foolbox_rewrite_spec`

rewrites

Examples

```
# This is a very simple inline function that require we
# provide the function body as it should be inserted.
# For a more detailed version, see the Tutorial vignette.
inline <- function(f, fn, body) {
   body <- substitute(body)
   rewrite(f) %>%
     rewrite_with(
         rewrite_callbacks() %>%
           add_call_callback(fn, function(expr, ...) body)
     )
}

g <- function(x) x**2
h <- rewrites[inline(g,y**2)] < function(y) y + g(y)
h
```

`add_call_callback` *Add a function-specific callback to the call callbacks.*

Description

This function adds to the existing call callback, rather than replace it, by putting a callback in front of it to be tested first. The callback will be invoked when the traversal sees a call to a specific function.

Usage

```
add_call_callback(callbacks, fn, cb)
```

Arguments

`callbacks`	The existing callbacks.
`fn`	The function to which calls should be modified.
`cb`	The callback function to invoke.

Details

The callback that is installed will be called with the usual callback arguments (which depend on context and user-provided information to ..., see `rewrite_callbacks()` and `analysis_callbacks()`), and additionally the next callback in line, through the parameter `next_cb`. This can be used to propagate information through several callbacks in a pipe-like fashion.

Value

`The updated callbacks.`

Examples

```
f <- function(x) 2 + x
cb <- rewrite_callbacks() %>%
   add_call_callback(f, function(expr, ...) {
       quote(2 + x)
   })
tr_f <- . %>% rewrite() %>% rewrite_with(cb)

g <- function(y) y + f(y)
tr_f(g)
```

---

add_topdown_callback *Add a function-specific callback to the top-down callbacks.*

Description

This function adds to the existing topdown callback, rather than replace it, by putting a callback in front of it to be tested first. The callback will be invoked when the traversal sees a call to a specific function.

Usage

```
add_topdown_callback(callbacks, fn, cb)
```

Arguments

callbacks	The existing callbacks.
fn	The function to which calls should be modified.
cb	The callback function to invoke.

Details

The callback that is installed will be called with the usual callback arguments (which depend on context and user-provided information to ..., see rewrite_callbacks() and analysis_callbacks()), and additionally the next callback in line, through the parameter next_cb. This can be used to propagate information through several callbacks in a pipe-like fashion.

Value

The updated callbacks.

annotate_assigned_symbols_callbacks

Put attribute "assigned_symbols" on expressions bottom-up

Description

Put attribute "assigned_symbols" on expressions bottom-up

Usage

```
annotate_assigned_symbols_callbacks
```

Format

An object of class `list` of length 7.

annotate_bound_symbols_in_function

Annotate sub-expressions with variables bound in their scope.

Description

Extracts all the symbols that appear on the left-hand side of an assignment or as function parameters and annotate each sub-expression with those.

Usage

```
annotate_bound_symbols_in_function(fn)
```

Arguments

`fn`	The function whose body we should analyse

Details

This function will annotate a function's body with two attributes for each sub-expression in the body. Each `call` expression in the body will be annotated with these two attributes:

- **assigned_symbols**: Variables that appear to the left of an assignment in a sub-expression of the call that is likely to affect the scope of the call.
- **bound**: Variables that are either assigned to, thus potentially local in the scope, or function parameters from an enclosing scope, which will definitely be bound at this position.

Since R does not require that we declare local variables, and since the variables that are assigned to a local scope depend on the runtime execution of functions, we cannot determine with any certainty which variables will be assigned to in any given scope at any given program point. So the best we can do is figure out which variables are *potentially* assigned to. Which is what this function does.

The rules for when we are assigning to a local variable are a bit complicated. For control structures, we can assume that assignments will be to the local scope. People can change the implementation of these so it isn't, but then they are only hurting themselves and deserve the extra pain we can give them. For other call arguments, it gets a little more complicated. With standard-evaluation, if we have an arrow assignment in a function argument, then the assignment happens in the calling scope. So we will assume this happens unless we are handling cases we know have NSE, such as `with`. If an assignment is inside a block, however, we will assume that NSE *is* in play, by default, and not consider it a local assignment.

Value

A function who's expressions are annotated with potentially local variables.

`annotate_bound_variables_callbacks`

Propagate parameters and local variables top-down to assign attribute "bound" to all call expressions.

Description

Propagate parameters and local variables top-down to assign attribute "bound" to all call expressions.

Usage

```
annotate_bound_variables_callbacks
```

Format

An object of class `list` of length 7.

`collect_assigned_symbols_in_expression`

Extracts all the symbols that appear on the left-hand side of an assignment.

Description

Since R does not require that we declare local variables, and since the variables that are assigned to a local scope depend on the runtime execution of functions, we cannot determine with any certainty which variables will be assigned to in any given scope at any given program point. So the best we can do is figure out which variables are *potentially* assigned to. Which is what this function does.

Usage

```
collect_assigned_symbols_in_expression(expr, env, params = list(),
  topdown = list())

collect_assigned_symbols_in_function(fun, topdown = list())
```

Arguments

expr	The expression to analyse
env	Environment in which to look up symbols.
params	Parameters for the function being analysed (if these are needed).
topdown	Information to pass top-down in the traversal.
fun	The function whose body we should analyse

Details

The collect_assigned_symbols_in_function() function reformats the collected data into a character vector, removes duplications, and remove the formal parameters of the function from the list, so those are not considered local variables (rather, they are considered formals and presumably handled elsewhere as such).

Value

A list containing the symbols that were assigned to.

Functions

- collect_assigned_symbols_in_expression: Analyse an expression.
- collect_assigned_symbols_in_function: Analyse the body of a function.

---

collect_from_args *Collect attributes set in the arguments to a call expression.*

Description

Given a call expression expr, this function scans the arguments to the call and extracts the attribute attribute from each where the condition predicate evaluates to TRUE, and it concatenates all these.

Usage

```
collect_from_args(expr, attribute, condition = function(expr) TRUE,
  include_fun = FALSE)
```

Arguments

expr	The expression to process.
attribute	The attribute we want to collect from the arguments.
condition	A predicate. Only arguments where the condition evaluates to TRUE will be included in the result.
include_fun	Include the first element in a call, i.e. the function that will be called.

Value

A list or vector obtained by concatenating the attributes from the arguments.

---

depth_first_analyse_expr

Analyse an expression.

Description

Traverses the expression expr depth-first and analyse it it using callbacks.

Usage

```
depth_first_analyse_expr(expr, callbacks, params = list(),
  topdown = list(), wflags = warning_flags(), ...)
```

Arguments

expr	An R expression
callbacks	List of callbacks to apply.
params	Parameters of the function we are analysing. If we are working on a raw expression, just use the default, which is an empty list.
topdown	A list of additional information gathered in the traversal.
wflags	Warning flags, see warning_flags().
...	Additional data that will be passed along to callbacks.

Value

The result of the last bottom-up traversal.

See Also

analysis_callbacks

identity_analysis_callback

depth_first_analyse_function

depth_first_analyse_function

Analyse the body of function.

Description

Traverses the body of `fn` and analyse it based on `callbacks`.

Usage

```
depth_first_analyse_function(fn, callbacks, topdown = list(),
  wflags = warning_flags(), ...)
```

Arguments

`fn`	A (closure) function.
`callbacks`	List of callbacks to apply.
`topdown`	A list of additional information that will be considered top-down in the traversal.
`wflags`	Warning flags, see `warning_flags()`.
`...`	Additional data that will be passed along to callbacks.

Value

The result of the last bottom-up call to a callback.

See Also

depth_first_analyse_expr

depth_first_rewrite_function

analysis_callbacks

depth_first_rewrite_expr

Transform an expression.

Description

Traverses the expression `expr` depth-first and transform it using `callbacks`.

Usage

```
depth_first_rewrite_expr(expr, callbacks, params = list(),
  topdown = list(), wflags = warning_flags(), ...)
```

Arguments

expr	An R expression
callbacks	List of callbacks to apply.
params	Parameters of the function we are rewriting. If we are working on a raw expression, just use the default, which is an empty list.
topdown	A list of additional information gathered in the traversal.
wflags	Warning flags, see warning_flags().
...	Additional data that will be passed along to callbacks.

Value

A modified expression.

See Also

rewrite_callbacks

identity_rewrite_callback

depth_first_rewrite_function

---

depth_first_rewrite_function

Transform the body of function.

Description

Traverses the body of fn and rewrite it based on callbacks.

Usage

```
depth_first_rewrite_function(fn, callbacks, topdown = list(),
  wflags = warning_flags(), ...)
```

Arguments

fn	A (closure) function.
callbacks	List of callbacks to apply.
topdown	A list of additional information that will be considered top-down in the traversal.
wflags	Warning flags, see warning_flags().
...	Additional data that will be passed along to callbacks.

Value

A new function similar to fn but with a transformed body.

See Also

depth_first_rewrite_expr

rewrite_callbacks

`identity_rewrite_callback` *A callback that does not do any transformation.*

Description

Callbacks have one required argument, `expr`, but will actually be called with more. The additional named parameters are:

Usage

```
identity_rewrite_callback(expr, ...)

identity_analysis_callback(expr, bottomup, ...)
```

Arguments

`expr`	The expression to (not) transform.
`...`	Additional named parameters.
`bottomup`	Information gathered depth-first in analysis callbacks. This parameter is only passed to callbacks in analysis traversals and not rewrite traversals.

Details

- **env** The function environment of the function we are transforming
- **params** The formal parameters of the function we are transforming
- **topdown** Data passed top-down in the traversal.
- **bottomup** Data collected by depth-first traversals before a callback is called. plus whatever the user provide to `depth_first_rewrite_function()` or `depth_first_analyse_function()`.
- **next_cb** A handle to call the next callback if more are installed. This variable will be the callback that was in the callbacks list before this one replaced it.

In bottom up analyses, the `merge_bottomup()` function can be used to collected the results of several recursive calls. When annotating expressions, the `collect_from_args()` can be used in `call` callbacks to extract annotation information from call arguments.

Value

`expr`

Functions

- `identity_rewrite_callback`: Identity for expression rewriting
- `identity_analysis_callback`: Identity for expression rewriting

See Also

merge_bottomup

collect_from_args

`make_with_callback` *Create a function for setting callbacks.*

Description

Create a function for setting callbacks.

Usage

```
make_with_callback(cb_name)
```

Arguments

`cb_name`	The name of the callback to set

Value

A function that can be used in a pipe to set a callback.

`merge_bottomup` *Merge the results of several bottomup results.*

Description

The `bottomup` parameter in callbacks will be calculated for all parameters of `call`` expressions. The parameter to the tomupparameter. If results are not named in thebottomup‘ list, they are discarded.

Usage

```
merge_bottomup(bottomup)
```

Arguments

`bottomup`	List of bottom up analysis results.

Details

The vectors from `bottomup` are concatenated, so one level of lists will be flattened. Use more lists, like `list(list(2), list(3))` if you want to prevent this.

See Also

depth_first_analyse_function

depth_first_analyse_expr

---

nop_topdown_callback *Top-down analysis callback.*

Description

Top-down analysis callback.

Usage

```
nop_topdown_callback(expr, topdown, skip, ...)
```

Arguments

`expr`	The expression before we modify it.
`topdown`	Information from further up the expression tree.
`skip`	An escape function. If called, the transformation or analysis traversal will skip this expression and continue at the sibling level.
`...`	Additional data that might be passed along

Value

Updated `topdown` information.

---

remove_formal *Remove a parameter from the formal parameters of a function.*

Description

Remove a parameter from the formal parameters of a function.

Usage

```
remove_formal(fn, par)
```

Arguments

fn	A function we are modifying
par	A parameter of fn (should be in formals(fn) and not be quoted)

Value

A modified function equal to fn but with par removed from the formal parameters.

---

remove_formal_ *Remove a parameter from the formal parameters of a function.*

Description

Remove a parameter from the formal parameters of a function.

Usage

```
remove_formal_(fn, par)
```

Arguments

fn	A function we are modifying
par	A parameter of fn (should be in formals(fn) and be quoted)

Value

A modified function equal to fn but with par removed from the formal parameters.

---

rewrites *Object for setting up a transformation pipeline when defining functions*

Description

Object for setting up a transformation pipeline when defining functions

Usage

```
rewrites
```

Format

An object of class foolbox_rewrite_spec of length 1.

Examples

```
# This is a very simple inline function that require we
# provide the function body as it should be inserted.
# For a more detailed version, see the Tutorial vignette.
# For a version that permits partial evaluation, see that vignette.
inline <- function(f, fn, body) {
   body <- substitute(body)
   rewrite(f) %>%
     rewrite_with(
         rewrite_callbacks() %>%
           add_call_callback(fn, function(expr, ...) body)
     )
}

g <- function(x) x**2
h <- rewrites[inline(g,y**2)] < function(y) y + g(y)
h
```

`rewrite_callbacks` *Default expression-transformation callbacks.*

Description

Callbacks must be functions that take three arguments: The expression to rewrite, the environment of the function we are rewriting (i.e. the environment it is defined in, not the function call frame), and a list of formal parameters of the function we are translating.

Usage

```
rewrite_callbacks()

analysis_callbacks()

with_atomic_callback(callbacks, fn)

with_pairlist_callback(callbacks, fn)

with_symbol_callback(callbacks, fn, include_missing = FALSE)

with_primitive_callback(callbacks, fn)

with_call_callback(callbacks, fn)

with_topdown_pairlist_callback(callbacks, fn)

with_topdown_call_callback(callbacks, fn)
```

Arguments

`callbacks`	The list of callbacks
`fn`	A function to install as a callback.
`include_missing`	For symbols, it is possible that the expression is missing. This can happen in pair-lists if a function parameter does not have a default argument. By default, the callback is not invoked on missing expressions–there is very little you can do with them – but you can include them by setting this parameter to `TRUE`.

Details

The flow of a depth-first traversal is as follows:

For expressions that are atomic, i.e. are either atomic values, pairlists, symbols, or primitives, the corresponding callback is called with the expression. The callbacks are called with the expression, `expr`, the environment of the function we are traversing, `env`, the parameters of that function, `params`, information collected top-down in `topdown`, warning flags through the `wflags` parameter, and any additional user-provided arguments through `...`. If the callbacks are used in a rewrite traversal, see `depth_first_rewrite_function()`, they must return an expression. This expression will be inserted as a substitute of the `expr` argument in the function being rewritten. If the callback is part of an analysis, see `depth_first_analyse_function()`, then it can return any data; what it returns will be provided to the callbacks on the enclosing expression via the `bottomup` parameter.

For `call` expressions, the `topdown` callback is invoked before the call is traversed. It is provided with the same arguments as the other callbacks and in addition a thunk `skip` that it can use to prevent the depth-first traversal to explore the call further. Whatever the `topdown` callback returns will be provided to the call callback via the argument `topdown` it it is called (i.e. if the `topdown` callback doesn't invoke `skip`).

After the `topdown` callback is executed, if it doesn't call `skip`, the `call` callback is called on the expression. It is called with the same arguments as the other callbacks, and must return an expression if part of a rewrite traversal or any collected information if part of an analysis traversal.

Functions

- `rewrite_callbacks`: Default callbacks for rewriting expressions
- `analysis_callbacks`: Default callbacks for analysing expressions
- `with_atomic_callback`: Set the atomic callback function.
- `with_pairlist_callback`: Set the pairlist callback function.
- `with_symbol_callback`: Set the symbol callback function.
- `with_primitive_callback`: Set the primitive callback function.
- `with_call_callback`: Set the call callback function.
- `with_topdown_pairlist_callback`: Set the topdown information-passing callback function for pair-lists
- `with_topdown_call_callback`: Set the topdown information-passing callback function for calls.

See Also

with_atomic_callback

with_symbol_callback

with_primitive_callback

with_pairlist_callback

with_call_callback

with_topdown_pairlist_callback

with_topdown_call_callback

warning_flags

Examples

```
f <- function(x) 2 + x
cb <- rewrite_callbacks() %>%
   add_call_callback(f, function(expr, ...) {
       quote(2 + x)
   })
tr_f <- . %>% rewrite() %>% rewrite_with(cb)

g <- function(y) y + f(y)
tr_f(g)

collect_symbols <- function(expr, ...) {
   list(symbols = as.character(expr))
}
callbacks <- analysis_callbacks() %>% with_symbol_callback(collect_symbols)
f %>% analyse() %>% analyse_with(callbacks)
```

`rewrite_with` *Functions for applying a sequence of rewrites.*

Description

The `rewrite()` function applies a series of transformations to an input function, `fn` and returns the result. This result can then be used in a pipeline of `rewrite_with()` calls for further analysis.

Usage

```
rewrite_with(fn, callbacks, ...)

rewrite(fn)

analyse(fn)

analyse_with(fn, callbacks, ...)
```

```
rewrite_expr(expr)

rewrite_expr_with(expr, callbacks, ...)

analyse_expr(expr)

analyse_expr_with(expr, callbacks, ...)
```

Arguments

fn	The function to rewrite
callbacks	The callbacks that should do the rewriting
...	Additional parameters passed along to the callbacks.
expr	When invoked on expressions, in rewrite_expr(), the expression to rewrite.

Details

The flow of transformations goes starts with rewrite() and is followed by a series of rewrite_with() for additional rewrite callbacks. For analysis, it starts with analyse() and is followed by a pipeline of analyse_with().

This functions will annotate a function's body with two attributes for each sub-expression in the body. Each call expression in the body will be annotated with these two attributes:

- **assigned_symbols**: Variables that appear to the left of an assignment in a sub-expression of the call that is likely to affect the scope of the call.
- **bound**: Variables that are either assigned to, thus potentially local in the scope, or function parameters from an enclosing scope, which will definitely be bound at this position.

Since R does not require that we declare local variables, and since the variables that are assigned to a local scope depend on the runtime execution of functions, we cannot determine with any certainty which variables will be assigned to in any given scope at any given program point. So the best we can do is figure out which variables are *potentially* assigned to. Which is what this function does.

The rules for when we are assigning to a local variable are a bit complicated. For control structures, we can assume that assignments will be to the local scope. People can change the implementation of these so it isn't, but then they are only hurting themselves and deserve the extra pain we can give them. For other call arguments, it gets a little more complicated. With standard-evaluation, if we have an arrow assignment in a function argument, then the assignment happens in the calling scope. So we will assume this happens unless we are handling cases we know have NSE, such as with. If an assignment is inside a block, however, we will assume that NSE *is* in play, by default, and not consider it a local assignment.

Value

A rewritten function

Functions

- `rewrite_with`: Apply `callbacks` over `fn` to rewrite it.
- `rewrite`: Function for starting a rewrite.
- `analyse`: Function for running analysis callbacks
- `analyse_with`: Apply `callbacks` over `fn` to analyse it.
- `rewrite_expr`: Expression version of `rewrite()`
- `rewrite_expr_with`: Expression version of `rewrite_with()`
- `analyse_expr`: Expression version of `analyse()`
- `analyse_expr_with`: Expression version of `analyse_with()`

See Also

rewrite_callbacks

Examples

```
f <- function(x) 2 + x
cb <- rewrite_callbacks() %>%
   add_call_callback(f, function(expr, ...) {
       quote(2 + x)
   })
tr_f <- . %>% rewrite() %>% rewrite_with(cb)

g <- function(y) y + f(y)
tr_f(g)

collect_symbols <- function(expr, ...) {
   list(symbols = as.character(expr))
}
callbacks <- analysis_callbacks() %>% with_symbol_callback(collect_symbols)
f %>% analyse() %>% analyse_with(callbacks)
```

---

warning_flags *Collection of warning flags used when traversing expressions.*

Description

These are flags for turning warnings on or off when traversing expression trees.

Usage

```
warning_flags()

set_warn_on_unknown_function(flags)

unset_warn_on_unknown_function(flags)

set_warn_on_local_function(flags)

unset_warn_on_local_function(flags)
```

Arguments

flags	Used when setting or unsetting flags.

Details

The flags can be provided to transformation and analysis functions, and be set or unset by the set_/unset_ functions. The meaning of the flags are:

- **warn_on_unknown_function:** If you have installed a callback with add_call_callback() or add_topdown_callback(), the traversal code will check if a given call is to a known function installed by one of these. If the function name of a call is not recognised as a function parameter or a local variable, as annotated with annotate_bound_symbols_in_function(), then the code will issue a warning if this flag is set. The warning behaviour depends on whether annotate_bound_symbols_in_function() has analysed the function. If it hasn't, then we only consider function parameters as local variables. If it has, we have more information about the local variables, so we can make the warnings more accurate. The flag is set by default.
- **warn_on_local_function:** If you have installed a callback with add_call_callback() or add_topdown_callback(), the traversal code will check if a given call is to a known function installed by one of these. If you have installed a function that has a name-clash with a local variable, and this flag is set, then you will get a warning. If you have annotated the expression tree using annotate_bound_symbols_in_function(), then the warning will be invoked both on local variables and function parameters; if you have not annotated the expression tree, then it will only be invoked on function arguments. The flag is set by default.

Since R is a very dynamic language, it is not possible to know which local variables might refer to functions and which do not – and R will look for functions if a variable is used as a call and potentially skip past a local variable that refers to a non-function – so the warnings are based on heuristics in identifying local variables and are conservative in the sense that they assume that if a call is to a name that matches a local variable, then it is the local variable that is being called.

Functions

- set_warn_on_unknown_function: Enable warnings when encountering an unknown function
- unset_warn_on_unknown_function: Disable warnings when encountering an unknown function

- `set_warn_on_local_function`: Enable warnings when encountering a local variable with a name that matches one installed for transformation.
- `unset_warn_on_local_function`: Disable warnings when encountering a local variable with a name that matches one installed for transformation.

`[.foolbox_rewrite_spec`

Provide list of rewrite transformations.

Description

This subscript operator is used together with `rewrites` to specify a sequence of transformations to apply to a new function we define.

Usage

```
## S3 method for class 'foolbox_rewrite_spec'
dummy[...]
```

Arguments

`dummy`	The dummy-table `rewrites`. It is only here because it allows us to use subscripts as part of the domain-specific language.
`...`	A list of rewrite functions.

See Also

`<.foolbox_pipe`

rewrites

Examples

```
# This is a very simple inline function that require we
# provide the function body as it should be inserted.
# For a more detailed version, see the Tutorial vignette.
inline <- function(f, fn, body) {
   body <- substitute(body)
   rewrite(f) %>%
     rewrite_with(
         rewrite_callbacks() %>%
           add_call_callback(fn, function(expr, ...) body)
     )
}

g <- function(x) x**2
h <- rewrites[inline(g,y**2)] < function(y) y + g(y)
h
```

Index