Package 'TypeInfo'

April 23, 2024

Version 1.69.0

Date 9/27/2005

Title Optional Type Specification Prototype

Author Duncan Temple Lang
Robert Gentleman (<rgentlem@fhcrc.org>)

Maintainer Duncan Temple Lang <duncan@wald.ucdavis.edu>

Description A prototype for a mechanism for specifying the types of parameters and the return value for an R function. This is meta-information that can be used to generate stubs for servers and various interfaces to these functions. Additionally, the arguments in a call to a typed function can be validated using the type specifications.
We allow types to be specified as either
i) by class name using either inheritance - is(x, className), or strict instance of - class(x) %in% className, or
ii) a dynamic test given as an R expression which is evaluated at run-time.
More precise information and interesting tests can be done via ii), but it is harder to use this information as meta-data as it requires more effort to interpret it and it is of course run-time information. It is typically more meaningful.

License BSD_2_clause

LazyLoad true

Depends methods

Suggests Biobase

Collate Classes.S checkArgs.S print.S rewrite.S support.S

biocViews Infrastructure

git_url https://git.bioconductor.org/packages/TypeInfo

git_branch devel

git_last_commit 4f5c77e

git_last_commit_date 2023-10-24

Repository Bioconductor 3.19

Date/Publication 2024-04-22

Contents

`checkArgs` *Validate the arguments in a call to a typed function.*

Description

`TypeInfo` uses `checkArgs` internally.

This function is used to validate the arguments in a call to a function that has associated type information about the parameters. The types for the parameters are currently given associated with the function via an attribute `"ParameterTypes"`. In the body of the function, one can call `checkArgs` and the specification is taken and used to compute whether the elements in the call are compatible with those in the signatures.

There are currently several ways to specify the signatures. One is as a list of explicit parameter name - class name pair vectors given as `c(paramName = className, paramName = className, ...)`. Alternatively, one can use an expression to perform a dynamic test. For example, one can test the length of an object, e.g. `c(x = length(x) < 4, y = length(y) == length(x))`. Each expression should return a logical value indicating whether the expected condition was satisfied. A third form of specifying signatures is given using class names for individual parameters and just matching the argument class to these names. This differs from the first form because the arguments are not checked simultaneoulsy, but rather one at a time. The test for a given argument is whether it is in the named vector of classes.

Usage

```
checkArgs(f = sys.function(1), argNames, args = NULL, forceAll = FALSE,
          env = sys.frame(1), isMissing = logical(0))
```

Arguments

f	the function object. If this is missing, the function is taken as the function being called in the previous frame, i.e. the one that called checkArgs.
argNames	a character vector giving the names of the arguments that are to be checked.
args	a list of named argument values.
forceAll	a logical value. If this is TRUE, then we evaluate all of the arguments in the call frame of the function being evaluated whose arguments we are to check. If this is FALSE, This should be a three-level enum to represent evaluate as needed, evaluate all referenced in any of the signatures and evaluate all of the arguments now.
env	the environment in which arguments are located.
isMissing	named logical vector indicating missing formal arguments; defined internally when consulting f of class function

.

Value

If the check succeeds in matching the arguments to the parameter types, the signature that matched is returned. Otherwise, an error is raised. If the signature is returned, this can be used to validate the return value in the context of that signature.

Note that if an instance of SimultaneousTypeSpecification-class is provided to this function, the TypedSignature-class elements are searched sequentially until a matching one is found. That matching signature is returned. Therefore, the order the signatures are specified within the SimultaneousTypeSpecification-class object is important. This could change if we wanted. At present, it is up to the author to specify what they want to have happen. We could use the S4 signature matching technique when this is finalized and implemented in C code.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

typeInfo

Examples

```
bob = function(x, y) {
        checkArgs()  # Completely unecessary as we don't specify type information.
        "Finished"
      }
```

```
 # a call generates a warning to say that there was no type information.
bob()
```

---

checkReturnValue *Verify the return value from the function has the appropriate type*

Description

This function is the counterpart to checkArgs in the type validation for an R function. When called, either implicitly or explicitly when the function returns, it attempts to determine whether the value being returned by the function call is valid relative to the type information of the function and the call itself. Specifically, it uses the signature of the current call to the function if it is available (returned by checkArgs) to see if it has a specified return type. If so, it compares the return value to that. Otherwise, it checks to see if the return type for the overall type info object (not just the specific type signature for the call) is specified and then uses that to validate the type. If neither is specified, then the value is not validated and the value returned.

Usage

```
checkReturnValue(returnType, returnJump, sig, f = sys.function(-1))
```

Arguments

returnType	the specifiedtype of the return value.
returnJump	this is a very special value which is a call to return the value of value. It must be explicitly given in the call to checkReturnValue and is used to ensure that the return from checkReturnValue returns from the calling function also in the case that the value is valid. This is a piece of magic in R that is very powerful using the lazy evaluation of the arguments that allows us to return from the place that the return call was specified.
sig	the signature corresponding to the call of the function f. This should have a returnType slot that contains class information or an expression. Otherwise, the value is taken from the TypeSpecification-class object for the entire function and its returnType slot.
f	the function object whose return value is to be validated. It is from this that we get the type info via typeInfo.

Value

If the validation takes place and is successful or simply doesn't take place because no returnType is available, the return value is value. Otherwise, if the validation fails, an error is raised.

Note

This is a prototype to illustrate the idea. It might be done in C code in the future within the R interpreter.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

checkArgs TypeSpecification-class

ClassNameOrExpression-class

Class "ClassNameOrExpression" to represent type information as either class names or arbitrary language test

---

Description

This class is used to represent a type test that is specified either as a collection of class names (and whether to check for strict equality or inheritance) or a dynamic predicate expression that is evaluated at run-time to determine whether the test is satisfied.

We may not need this in the "new" class hierarchy. It was created originally to be a union of character vectors, calls or expressions. But now that we have NamedTypeTest and DynamicTypeTest, we could perhaps use a common base class.

Objects from the Class

A virtual Class: No objects may be created from it.

Methods

No methods defined with class "ClassNameOrExpression" in the signature.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

TypedSignature TypedSignature-class ReturnTypeSpecification

DynamicTypeTest-class *Class "DynamicTypeTest" for computed tests on objects.*

Description

This virtual class is used interntally to unite type signatures that perform a computation to assess argument type suitability.

Slots

None.

Extends

Class "ClassNameOrExpression", directly.

Methods

None.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

TypedSignature TypeSpecification-class

Examples

```
checkedSqrt <- function(x) {
    return(sqrt(x))
}

typeInfo(checkedSqrt) <-
  SimultaneousTypeSpecification(
    TypedSignature(x=quote(
      is(x, "numeric") && all(x>=0))))

typeInfo(checkedSqrt)

checkedSqrt(2)
try(checkedSqrt(-2))
```

hasParameterType *Functions to query existence of specific type information*

Description

These functions and the associated methods are used to determine if a function has type specification for any of the parameters and also for the return type. These are used when rewriting the body of the function to support type checking (see rewriteTypeCheck). We use these predicate functions to determine if we have information about any parameter types and if not we do not add a check of the arguments (i.e. a call to checkArgs). Similarly, we determine if we have any information about the return type before adding a call to checkReturnValue.

They are used internally. They are exported in order to make them available for others to use in providing alternatives to this prototype implementation and also to overcome an anomoly in the callNextMethod() mechanism that appears to disappear when the generic is exported from the NAMESPACE.

Usage

```
hasParameterType(def)
```

Arguments

def	the object which is to be queried. This can be a function or a TypeSpecification-class instance which is typically extracted from the function. Generally, a user would pass the function to the function call and the resulting sequence of recursive method calls will occur.

Value

A logical value indicating if the object def "has" the relevant facet/property.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

rewriteTypeCheck checkArgs checkReturnValue

Examples

```
hasReturnType(SimultaneousTypeSpecification(
                      TypedSignature(x="integer", returnType = "duncan")))


# FALSE
hasReturnType(SimultaneousTypeSpecification(TypedSignature(x="integer")))
```

```
# TRUE
hasReturnType(SimultaneousTypeSpecification(returnType = "duncan"))

# TRUE
hasReturnType(ReturnTypeSpecification("duncan"))

hasReturnType(IndependentTypeSpecification(x = c("integer", "logical"),
                                           y  = "character",
                                           returnType = "duncan"))

hasReturnType(IndependentTypeSpecification(x = c("integer", "logical"),
                                           y  = "character"))
```

IndependentTypeSpecification

Create separate type information for different parameters.

Description

This function is a constructor for the IndependentTypeSpecification-class class. In short, it collects information about the possible types of parameters that is used to validate arguments in a call separately. This contrasts with checking the combination of arguments in the call against a particular signature.

Usage

```
IndependentTypeSpecification(..., returnType, obj = new("IndependentTypeSpecification", list(...)))
```

Arguments

...	name elements of which are either character vectors or expressions/calls that can be evaluated. These are of type ClassNameOrExpression-class.
returnType	the expected type of the return value. This is optional.
obj	the instance of class TypeSpecification-class that is to be populated with the values from ... and returnType.

Value

The return value is obj after it has been populated with the arguments ... and returnType.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

typeInfo, typeInfo<- checkArgs, checkReturnValue IndependentTypeSpecification-class SimultaneousTypeSpecification SimultaneousTypeSpecification-class

Examples

```
pow = function(a, b)
{
     # the return here is important to ensure the return value is checked.
   return(a^b)
}

typeInfo(pow) =
  IndependentTypeSpecification(
        a = c("numeric", "matrix", "array"),
        b = "numeric",
        returnType = quote(class(a))
   )


IndependentTypeSpecification(
        a = c("numeric", "matrix", "array"),
        b = new("StrictIsTypeTest","numeric"),
        c = new("StrictIsTypeTest",c("numeric", "complex")),
        d = as("numeric", "NamedTypeTest"),
        e = new("InheritsTypeTest", c("numeric", "complex"))
 )
```

`NamedTypeTest-class` *Class "NamedTypeTest" and sub-classes for tests on class of an object.*

Description

These classes are for specifying a test on the type of an object using the class of that object and comparing it to target class names. The tests can be either for `x inherits from class name` (or `is(x, "className")`) or `x is an instance of class name` (i.e. `class(x) == "className"`). The first of these is represented by `InheritsTypeTest` and the second by `StrictTypeTest`.

Objects from the NamedTypeTest class

Objects can be created for the non-virtual classes using `new("InheritsTypeTest",...)` and `new("StrictIsTypeTest",..` or the convenience functions `InheritsTypeTest(...)`, `StrictIsTypeTest(...)` Additionally, where appropriate, a character vector is coerced to `InheritsTypeTest`.

Slots

`.Data`**:** Object of class `"character"`. This is an internal data type to represent the class names. It is not to be used directly. It is inherited from the "character" class.

Extends

Class `"character"`, from data part. Class `"ClassNameOrExpression"`, directly. Class `"vector"`, by class `"character"`.

Methods

coerce `signature(from = "character", to = "NamedTypeTest")`: converts a character vector into a `InheritsTypeTest`.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

`TypedSignature` `TypeSpecification-class` `DynamicTypeTest-class`

Examples

```
new("InheritsTypeTest",  c("A", "B"))

m = array(1:60, c(3, 4, 5))
tt = new("StrictIsTypeTest",  c("matrix"))
TypeInfo:::checkType(m, tt)

tt = new("StrictIsTypeTest",  c("array"))
TypeInfo:::checkType(m, tt)
```

---

paramNames — *Compute the names of all the specified parameters in a TypeSpecification object*

Description

This generic function has methods for computing the names of all the parameters in a type specification for a function for which type information is explicitly specified. There is also a method for functions which merely returns the names of the formal parameters, i.e. a call to `formals`.

Usage

```
paramNames(def)
```

Arguments

`def`	the object from which we are to identify the names of the parameters

Value

A character vector giving the names of the parameters that were found.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

typeInfo

ReturnTypeSpecification *Constructor for specifying information about only the return type*

Description

This function is a constructor for a class that represents information only about the return type of a function and explicitly has no information about the parameters.

Usage

```
ReturnTypeSpecification(type, obj = new("ReturnTypeSpecification"))
```

Arguments

type	the type specification. This should be an object of class ClassNameOrExpression or coercible to one.
obj	the instance that is to be populated and returned.

Value

By default, an object of class ReturnTypeSpecification-class. However, it merely returns the value of obj after populating it with the value of type. So strictly the return value is the augmented value of obj.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

IndependentTypeSpecification SimultaneousTypeSpecification

Examples

```
ReturnTypeSpecification(quote(length(x) == 3))

ReturnTypeSpecification("matrix")

ReturnTypeSpecification(new("StrictIsTypeTest", "matrix"))
```

`rewriteTypeCheck` *Insert code to support type validation*

Description

This generic function and its methods are used to modify the expressions in the body of a function in order to support the validation of type information in calls to this function. This changes the form of explicit calls to `return`, modifies the last expression if it is not an explicit call to return, and adds an initial command to compute check the arguments in the cal via `checkArgs`.

Usage

```
rewriteTypeCheck(f, doReturn = TRUE, checkArgs = TRUE, addInvisible = FALSE)
```

Arguments

`f`	the object which is to be modified to add the information for checking the return value and checking the input arguments. These are functions, expressions, calls, and other language objects.
`doReturn`	a logical value. If this is `FALSE`, the modifications are greatly simplified and no additions are made to handle the validation of the return value. This is used when the type information provides no information about the return type and so it cannot be validated or constrained.
`checkArgs`	a logical value indicating whether the modifications should including check the arguments. If the only type information given is about the return type, no checking of the arguments is necessary (in the current model).
`addInvisible`	logical indicating whether returned argument needs to be cloaked in `invisible`.

Value

The potentially modified version of the original input argument. The modifications contain any necessary changes to support the type checking at run-time.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

`typeInfo` `checkArgs` `checkReturnValue`

Examples

```
f = function(x, y) {
  z = x + y
  sum(z)
}
```

showTypeInfo *Display information about argument types*

Description

This generic function returns type specifications as a list. Elements in the list contain information about different parts of the signature. The order and white spece of the list suggests structure of the type specification.

showTypeInfo is usually invoked with a single argument, the name of the function with type information.

Usage

```
showTypeInfo(object, name=character(), prefix="", ...)
```

Arguments

object	The object about which type information is required
name	Class name, not normally specified by user.
prefix	Used by methods to ensure pretty indentation type specifications.
...	Additional arguments used for derivatives of NamedTypeTest, not noramlly assigned by user.

Value

A list containing type information for def

Author(s)

MT Morgan <mtmorgan@fhcrc.org>

See Also

typeInfo

Examples

```
foo <- function(x) { return(x) }
typeInfo( foo ) <- SimultaneousTypeSpecification(
  TypedSignature( x = "numeric" ),
  returnType = "numeric" )
res <- showTypeInfo( foo )
cat( res, sep="\n" )
```

SimultaneousTypeSpecification *Create type signature information governing parameters in a call.*

Description

This function is a constructor for specifying different permissible combinations of argument types in a call to a function. Each combination of types identifies a signature and in a call, the types of the arguments are compared with these types. If all are compatible with the specification, then the call is valid. Otherwise, we check other permissible combinations.

Note that if an instance of SimultaneousTypeSpecification-class is provided to the checkArgs function, the TypedSignature-class elements are searched sequentially until a matching one is found. That matching signature is returned. Therefore, the order the signatures are specified within the SimultaneousTypeSpecification-class object is important. This could change if we wanted. At present, it is up to the author to specify what they want to have happen. We could use the S4 signature matching technique when this is finalized and implemented in C code.

Usage

```
SimultaneousTypeSpecification(..., returnType, obj = new("SimultaneousTypeSpecification", list(...)))
```

Arguments

...	named TypedSignature objects. The names identify the parameter to which the type specification applies.
returnType	if supplied this should be an object of class ClassNameOrExpression-class.
obj	the instance of TypeSpecification-class that is to be populated with the content of ... and returnType.

Value

The return value is obj. By default, this has class SimultaneousTypeSpecification-class. It should be an object of class TypeSpecification-class.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

IndependentTypeSpecification typeInfo

Examples

```
foo =
function(x, y)
{
  x + y
}

typeInfo(foo) =
    SimultaneousTypeSpecification(
         TypedSignature(x = "integer", y = "integer"),
         TypedSignature(x = "numeric", y = "logical"))
```

`TypedSignature` *Constructor for a TypedSignature object*

Description

This is a constructor function for the `TypedSignature-class` that represents constraints on the types or values of a combination of parameters. It takes named arguments that identify the types of the parameters. Each parameter type should be an object that is "compatible with" `ClassNameOrExpression-class`, i.e. a test for inheritance or a dynamic expression.

Usage

```
TypedSignature(..., returnType, obj = new("TypedSignature", list(...)))
```

Arguments

`...`	the types for the parameters given as `name = type` to identify the parameter and its type description.
`returnType`	the type description for the return value. This applies to the particular combination of inputs given in ...
`obj`	the instance to populate with the information given in the other arguments. This allows us to pass in objects of sub-classes to this function or to populate previously created objects.

Value

The populated value of `obj`, by default an object of class `TypedSignature-class`.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

`SimultaneousTypeSpecification` `typeInfo` `checkArgs`

Examples

```
TypedSignature(x = "logical", y = quote(length(y) == length(x)))
```

`TypedSignature-class` *Class "TypedSignature" representing type information about function parameters and the return type*

Description

This class is used to describe an AND or simultaneous condition on the types of several arguments of a function call. The entire test is satisfied if all the individual elements are satisfied. One can represent the test elements for the different parameters as either class names (i.e. character strings or `NamedTypeTest-class` and sub-classes), and also predicate expressions using `DynamicTypeTest`.

In addition to the types on the parameters, one can also specify a test for the return type if a call to the function matches this signature. This allows us to associate a specific return type with a specific set of input types.

Currently this class is only used to describe the elements in `SimultaneousTypeSpecification-class` objects.

Objects from the Class

Use the constructor function `TypedSignature` to create objects of this class.

Slots

`.Data`**:** This object extends list. But this slot is intended to be opaque and should not be used directly.

`returnType`**:** Object of class `"ClassNameOrExpression"`. This represents the description of the return type of the function associated with this set of given input types.

Extends

Class `"list"`, from data part. Class `"vector"`, by class `"list"`.

Methods

hasParameterType `signature(def = "TypedSignature")`:

hasReturnType `signature(def = "TypedSignature")`:

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

`SimultaneousTypeSpecification`

typeInfo *Get or set type information for a function.*

Description

These functions provide controlled access to type information for a function. They encapsulate the way the information is stored for the function (although it is trivial to find out how it is done and where the information is)

Usage

```
typeInfo(func)
"typeInfo<-"(func, rewrite = TRUE, value)
```

Arguments

func	the function whose type information is to be accessed
rewrite	a logical value. This controls whether the body of the function func is rewritten so that checks for the arguments and return type are added to the code and the appropriate actions are taken when the function returns control to the caller. This is necessary to get calls to return in the function to behave correctly and allow the return value to be validated.
value	an object of class TypeSpecification-class

Value

typeInfo returns the type information associated with the function. typeInfo<- returns the modified function.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://www.omegahat.org/OptionalTyping

See Also

checkArgs checkReturnValue

TypeSpecification-class

Class "TypeSpecification" and derived class

Description

The classes in this collection are used to represent type information about a function in different ways. TypeSpecification is the virtual base class and provides the common slot to describe the type for the return value of the function.

The ReturnTypeSpecification-class is used when there is no information about the parameters of the function (either because there are no parameters or because we have no constraints on them).

The classes IndependentTypeSpecification-class and SimultaneousTypeSpecification-class are used to describe constraints on the arguments to the function. Both are lists, but behave very differently in the type checking. The difference is more difficult to describe succinctly than it is conceptually.

SimultaneousTypeSpecification-class is used when we want to specify information about the types of several arguments in a call taken as a group and imposing a constraint on that group of values. This corresponds to a call signature in the method dispatching. It says match each argument in turn with the given types and confirm the match over all of these tests. For example, we might have a function that accepts either (a) two numbers, or (b) two matrices. In that case, we need to specify the acceptable argument types as pairs: c("numeric", "numeric") and c("matrix", "matrix"). The key idea here is that the constraints on the types are AND-ed together across the different arguments. In our example, we impose the constraint `is.numeric(arg1) && is.numeric(arg2)`.

The IndependentTypeSpecification-class is used when we want to specify something about the types of different parameters but do not want the types to be AND-ed together. If we had a function that accepts a matrix or a number for its first parameter, and a matrix or string for its second parameter or any combination of those, then we would use the IndependentTypeSpecification-class. The term 'independent' is intended to suggest that the type checking is done for each parameter separately or independently of the others and then the check succeeds if all arguments pass. The phrase simultaneous means that we test the types of the arguments as a unit or simultaneously. The names can be easily changed to something more suggestive. It is the concept that is important.

A description of a quite different nature may also help and also provide information about the contents of these different list classes. For IndependentTypeSpecification-class, one can think of the list as having an element for each parameter for which we want to specify type information. This element is, at its simplest, a character vector giving the names of the acceptable classes. (We can have more complex elements such as expressions.) I think of this as being a collection of column vectors hanging from the parameters.

For SimultaneousTypeSpecification-class, we have rows or tuples of type information. These are call signatures. So we have

IndependentTypeSpecification corresponds to the SimultaneousTypeSpecification in the following computational manner. We can take the cartesian product (e.g. via expand.grid) of the inputs for IndependentTypeSpecification to form all possible combinations of types for the parameters and then we have the tuples for the corresponding SimultaneousTypeSpecification.

Constructors

One can create objects of the three non-virtual classes using the corresponding constructor functions in the package. These are ReturnTypeSpecification, IndependentTypeSpecification, SimultaneousTypeSpecification.

Slots

`.Data`: each of the non-virtual classes is really a list. They inherit the list properties and all the relevant methods. This slot is implementation specific and should not be used.

`returnType`: Object of class ClassNameOrExpression-class. This describes the return type for the function. In `SimultaneousTypeDescription` objects, we can also specify return type information corresponding to each signature, i.e. in the TypedSignature-class.

Extends

Class `"list"`, from data part. Class `"TypeSpecification"`, directly. Class `"vector"`, by class `"list"`.

Methods

Available methods are computed in the example below; see the corresponding help page for details.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

IndependentTypeSpecification SimultaneousTypeSpecification ReturnTypeSpecification typeInfo, typeInfo<- checkArgs, checkReturnValue

Examples

```
showMethods(classes=c(
              "TypeSpecification",
              "IndependentTypeSpecification",
              "SimultaneousTypeSpecification",
              "ReturnTypeSpecification"))
```

Index