Type specification for your functions

by MT Morgan, Seth Falcon, Robert Gentleman, and Duncan Temple Lang

You've written some amazing R functions. How can others, even non-R users, benefit from your hard work? Maybe you can make it easy for other programmers to learn about the arguments and return values of your function? Perhaps a web-based form or dialog box, like those provided by **widgetInvoke**, would help users choose appropriate arguments? These objectives are easier to obtain when R functions provide information about themselves.

The **TypeInfo** package annotates functions with information about argument and return types. **TypeInfo** automatically checks that your function is called with appropriate arguments. You can then focus on writing the code in the body of your function, rather than checking values supplied by users. Other R programmers can ask functions about their argument and return types. This 'reflection' opens the door to creative possibilities, for instance automatically creating a work flow (perhaps a graphical 'wizard'?) chaining function calls together into a complicated overall analysis.

This article illustrate how to use **TypeInfo** to specify argument and return types. We start with straight-forward ways of applying type information. Then we illustrate the flexibility of **TypeInfo** for apply complicated type checks, including types satisfying arbitrary R expressions. The article concludes with a brief look behind the scenes to expose limitations of **TypeInfo**, and to highlight opportunities for using typed functions in advanced aspects of your own work.

The basics: applying `typeInfo`

Suppose your colleagues clamor for a function to perform one-way analysis of variance on data where the predictor is a `factor`. Easily done in R with `lm`, but the R formula notation might be more than needed for our simple function. To help our colleagues, we simplify the interface to refer to `response` and `predictor` variables. Many users expect an ANOVA table as output, so we return the result of `anova` rather than `lm`. Here is our function:

```
> oneWayAnova <- function( response, predictor ) {
+     expr <- substitute( response ~ predictor )
+     result <- lm( as.formula( expr ))
+     anova( result )
+ }
> copyOfOneWayAnova <- oneWayAnova
```

We make a copy of the function definition to conveniently re-apply different type information, as will become apparent below. To test our function, we use data from the help page for `lm`:

```
> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33
> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32
> group <- gl(2,10,20, labels=c("Ctl","Trt"))
> weight <- c(ctl, trt)
> oneWayAnova( weight, group )

Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1 0.6882 0.68820  1.4191  0.249
Residuals 18 8.7292 0.48496
```

Applying type checks

We want to make sure our users enter the right kinds of arguments. To do this, we add type information to make sure that `response` is `numeric`, and `predictor` a `factor`. We start by loading the **TypeInfo** library...

```
> library(TypeInfo)
```

and, after defining `oneWayAnova`, add type information:

```
> typeInfo(oneWayAnova) <-
+   SimultaneousTypeSpecification(
+     TypedSignature(
+       response = "numeric",
+       predictor = "factor"),
+     returnType = "anova")
```

The command `SimultaneousTypeSpecification` means that we want to impose a set of conditions that apply simultaneously to all arguments (and, optionally, the return type). `TypedSignature` corresponds to one set of conditions. The structure of `TypedSignature` is a list of argument names and their types. Specifying `returnType` advertises that our function returns an object of type `anova`, and checks that it really does return this type. The reason for placing `returnType` after `TypedSignature` is clarified below.

Using a typed function is exactly the same as using an untyped function:

```
> oneWayAnova( weight, group )

Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1 0.6882 0.68820  1.4191  0.249
Residuals 18 8.7292 0.48496
```

```
> typeInfo(oneWayAnova)

An object of class "SimultaneousTypeSpecification"
[[1]]
An object of class "TypedSignature"
[[1]]
An object of class "InheritsTypeTest"
[1] "numeric"

[[2]]
An object of class "InheritsTypeTest"
[1] "factor"

Slot "returnType":
An object of class "InheritsTypeTest"
[1] NA


Slot "returnType":
An object of class "InheritsTypeTest"
[1] "anova"

> ngroup <- as.numeric( group )
> res <-
+     tryCatch(oneWayAnova( weight, ngroup ),
+              error=function(err) {
+                  cat("Error:",
+                      conditionMessage(err), "\n")
+              })

Error: TypeInfo could not match signature.
Supplied arguments and their types:
  response: numeric
  predictor: numeric
Available signature(s):
  [SimultaneousTypeSpecification]
    [TypedSignature]
      response: is(response, c('numeric'))  [InheritsTypeTest]
      predictor: is(predictor, c('factor'))  [InheritsTypeTest]
    returnType: is(returnType, c('anova'))  [InheritsTypeTest]
```

Figure 1: Finding out about type information, and the informative consequences of supplying incorrect arguments.

Finding out about type information

Once applied, functions can be queried with `typeInfo`.

```
> typeInfo(oneWayAnova)
```

The output, in Figure 1, illustrates how type information is stored. `typeInfo` returns an object of class `SimultaneousTypeSpecification`. The object contains a list of objects of class `TypedSignature`, and a `returnType` slot. Each `TypedSignature` is a list with entries for each element with type specification.

The information returned from `typeInfo` provides useful information as-is. It can also be parsed by computer code to provide information useful in creation of graphical widgets or other interfaces.

Incorrect arguments

When the user supplies incorrect data, e.g., representing the `predictor` as `numeric` rather than `factor`

```
> ngroup <- as.numeric( group )
> res <- try( oneWayAnova( weight, ngroup ))
```

TypeInfo intervenes with the error show in Figure 1. Providing **TypeInfo** is helpful, as our user might otherwise have performed a linear regression rather than fixed-effects ANOVA.

Elaborating on type signatures

We might decide that, for our purposes, the `predictor` can be either `factor` or `numeric`. We change the `SimultaneousTypeSpecification` to include another `TypedSignature` (re-applying `typeInfo` does *not* automatically overwrite previous type specifications, so we must use `typeInfo` on a fresh version of `oneWayAnova`):

```
> oneWayAnova <- copyOfOneWayAnova
> typeInfo(oneWayAnova) <-
+   SimultaneousTypeSpecification(
+     TypedSignature(
+       response = "numeric",
+       predictor = "factor"),
+     TypedSignature(
+       response = "numeric",
+       predictor = "numeric"),
+     returnType = "anova")
```

This starts to show the flexibility of **TypeInfo**. `SimultaneousTypeSpecification` allows for more than one `TypedSignature`. At least one of the `TypedSignatures` must be correct for the function to be evaluated. Conceptually, `SimultaneousTypeSpecification` performs a logical OR operation across the `TypedSignatures`. On the other hand, each `TypedSignature` specifies conditions that must all apply. `TypedSignature` performs a logical AND on its elements. In the example here, regardless of argument type, the function returns an object of class `anova`.

Flexible TypeInfo

The presentation so far emphasizes the sort of basic type specification that is likely to be most useful when making R functions available to other programming languages. **TypeInfo** offers a range of methods for validating type that can be very useful for R programmers, but that employ concepts not readily translated into other languages. A sampling of these are presented here, along with additional detail about the application of type specification .

`InheritsTypeTest`

Notice in the example above that arguments are labeled with character strings of type names. A type specification of class `character` corresponds to an `InheritsTypeTest`, as indicated explicitly for the `returnType` specification. An `InheritsTypeTest` requires that the object belong to, or extends, the specified class. For instance, the values passed to the function in the `response` variable must return `TRUE` from the test `is(response, "numeric")`. Because of this, `oneWayAnova` works with `response` as either `numeric` or `integer`.

```
> iweight <- as.integer( weight )
> oneWayAnova( iweight, group )

Analysis of Variance Table

Response: iweight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1    1.8 1.80000  2.9455 0.1033
Residuals 18   11.0 0.61111
```

`StrictIsTypeTest` and `DynamicTypeTest`

What other ways does **TypeInfo** offer to specify type? `StrictIsTypeTest` requires an exact match between the class of an object and the specified class(es). To specifying a strict match for `response` and `returnValue`, but an inherited match for `predictor`, write

```
> oneWayAnova <- copyOfOneWayAnova
> typeInfo(oneWayAnova) <-
+   SimultaneousTypeSpecification(
+     TypedSignature(
+       response = StrictIsTypeTest("numeric"),
+       predictor = InheritsTypeTest("factor")),
+     returnType = StrictIsTypeTest("anova"))
> oneWayAnova(iweight, group) # ERROR
```

Both StrictIsTypeTestand InheritsTypeTest accept a vector of class names.

The DynamicTypeTest allows evaluation of arbitrary expressions during type checking. For instance, the ANOVA anticipates that the length of predictor is the same as the length of response:

```
> typeInfo(oneWayAnova) <-
+   SimultaneousTypeSpecification(
+     TypedSignature(
+       response = "numeric",
+       predictor = quote(
+         length(predictor) ==
+             length(response) &&
+         is(predictor, "factor"))),
+     returnType = StrictIsTypeTest("anova"))
> short <- weight[-1]
> oneWayAnova(short, group)   # ERROR
```

Note that the expression in DynamicTypeTest has access to argument names, and uses quote to protect premature evaluation. DynamicTypeTest can also be used in the return statement.

Return types

As written here, the returnType applies to all TypedSignature's. That is, the function always returns an anova object, regardless of argument type. Actually, each TypedSignature can have its own returnType, allowing for a return type that depends on argument type.

IndependentTypeSpecification

We saw how several TypedSignature statements allow different types for predictor. IndependentTypeSpecification provides another mechanism to specifying alternative types:

```
> oneWayAnova <- copyOfOneWayAnova
> typeInfo(oneWayAnova) <-
+   IndependentTypeSpecification(
+     response = "numeric",
+     predictor = c( "factor", "numeric" ))
```

IndependentTypeSpecification expects a list of argument names, each with a character vector of possible types. While SimultaneousTypeSpecification performs a logical OR across each TypedSignature, IndependentTypeSpecification performs logical OR within arguments.

Behind the scenes: TypeInfo limitations and opportunities

Several aspects of **TypeInfo** provide opportunities for creative application. Type tests in **TypeInfo** form a hierarchy of S4 classes. This makes it easy to transform typeInfo output to structures or text representations that interface with other packages or programming languages. The approach is to specify methods that traverse the **TypeInfo** hierarchy, transforming **TypeInfo** objects into the desired format. InheritsTypeTest and StrictIsTypeTest rely on named classes, including user-defined S4 classes. Coupled with methods for parsing S4 objects (e.g., in the **XML** package), this provides one route to automatically generating **widgetInvoke** graphical interfaces or bindings for web-based services.

TypeInfo is not a universal solution. Perhaps the biggest limitation is that **TypeInfo** does not deal with S4 methods. The rationale is that arguments used for method dispatch must already satisfy type criteria. However, S4 methods may contain arguments that are not used for method dispatch, and the type of these arguments cannot be typed. S4 method dispatch imposes a test equivalent only to SimultaneousTypeSpecification in conjunction with InheritsTypeTest, rather than allowing the flexibility of **TypeInfo**. In addition, the valueClass argument of setMethod allows specification but not checking of the return type. A useful extension would enable typeInfo to query S4 generics, providing a uniform interface to retrieving type information.

Conceptually, **TypeInfo** works by inserting type-checking code at the first line of the function, and after (implicit or explicit) return statements. The extra code requires evaluation, slowing execution and making **TypeInfo** inappropriate in situations where speed is of the essence. In such situation, present a public type-checked 'wrapper' to help ensures only correct argument types reach the speed-critical functions. **TypeInfo** does not usually have side-effects, but a poorly written DynamicTypeTest could alter argument or return values unexpectedly.

Final hints and tips

We have seen how **TypeInfo** provides reflection, annotating function definitions with information to check argument types. The reflection provided by **TypeInfo** has several advantages. The user is informed of incorrect arguments, avoiding possibly subtle errors during function execution. The programmer is free to focus on the body of the function, rather than argument type checking. Other programmers can determine argument requirements or return values without detailed inspection of code or documentation. These and other advantages suggest application of **TypeInfo** as a way to enhance the effectiveness of your R programming.

For many purposes, the combination of SimultaneousTypeSpecification, TypedSignature, and a single returnType is the best way to use **TypeInfo**. This results in type signatures that translate readily into prototype concepts in other programming languages, making it easier for both R and non-R users to benefit from the functions you create. More elaborate formulations, especially `DynamicTypeTest`, are unlikely to be useful outside the R community, and may unnecessarily blur the distinction between type information, argument validation, and function execution. On the other hand, clearly specified argument types may aid rigorous formal testing (e.g., unit and regression tests) of return values prior to package release.

TypeInfo can be useful for all functions, but is especially beneficial for functions exported in a package NAMESPACE. Don't forget to add **TypeInfo** to the 'Depends' field of your package DESCRIPTION file!