



— **@:annotations** —

---

---

Aviral Goel, Jan Vitek, Petr Maj

# Use-case: Argument checking

```
mean.default <- function(x, trim = 0, na.rm = FALSE, ...) {  
  if(!is.numeric(x) &&!is.complex(x) &&!is.logical(x)) {  
    warning("...not numeric...")  
    return(NA)  
  }  
  if (na.rm) x <- x[!is.na(x)]  
  if(!is.numeric(trim) || length(trim) != 1L)  
    stop("'...numeric...length 1")  
  n <- length(x)  
  if(trim > 0 && n) {  
    if(is.complex(x))  
      stop("...not defined for complex data")  
    ...  
  }  
  .Internal(mean(x))  
}
```

# Use-case: Argument checking

```
mean.default <- function(???) {  
  if (na.rm) x <- x[!is.na(x)]  
  n <- length(x)  
  if(trim > 0 && n) {  
    ...  
  }  
  .Internal(mean(x))  
}
```

# Use-case: Argument checking

```
mean.default <- function(@:numeric[]|logical[]|complex[] x,  
                        @:!numeric trim = 0,  
                        @:logical na.rm = FALSE,  
                        ...)  
  
  @:numeric {  
    if (na.rm) x <- x[!is.na(x)]  
    n <- length(x)  
    if(trim > 0 && n) {  
      ...  
    }  
    .Internal(mean(x))  
  }  
}
```

# Use-case: Argument checking

```
mean.default <- function(@:numeric[]|logical[]|complex[] x,  
                          @:!numeric trim = 0,  
                          @:logical na.rm = FALSE,  
                          ...)  
  
  @:numeric  
  @:contract(trim>0 && length(x) && !is.complex(x)) {  
    if (na.rm) x <- x[!is.na(x)]  
    n <- length(x)  
    .Internal(mean(x))  
  }  
}
```

# Use-case: Expressions

```
{  
  # type annotation for assignment  
  @:integer x <- max(10, y)  
  # this should add profiling instructions to f  
  @:profile f()  
  # check that x is an integer vector of dim 3x3 convert if not  
  12 + @:integer(Warn,Conv) [3,3] x  
}
```

# Use-case: Documentation and ...

```
add_to_database <-  
  @:creator("John", "Gee", "jg@r.com")  
  @:description("Adds an entry to the database.")  
  @:version(1.1)  
  @:export  
  @:suppressMessage("*log*")  
  @:logErrors  
  @:authenticate  
  function(@:character id, @:numeric salary) { ... }
```

# Proof-of-concept

<https://github.com/aviralg/r-3.4.0>

Supports annotations on function , function formals and function body.

Annotations are arbitrary R expressions.

Annotations are just the expression AST.

Attached as attributes by the parser to the function's body.

A minimal API to get and set annotations.

179	src/main/gram.y
67	src/main/eval.c
2	src/include/names.c
1	src/main/Defn.h



# Proof-of-concept

```
> adder <-  
  @:export  
  function (@:integer|double i,  
           @:integer|double j)  
    @:integer|double {  
      i + j  
    }
```

```
> annotations(adder, "header")  
[[1]]  
export
```

# Proof-of-concept

```
> adder <-  
  @:export  
  function (@:integer|double i,  
           @:integer|double j)  
    @:integer|double {  
      i + j  
    }
```

```
> annotations(adder, "formals")
```

```
$i  
$i[[1]]  
integer | double
```

```
$j  
$j[[1]]  
integer | double
```

# Proof-of-concept

```
> adder <-  
  @:export  
  function (@:integer|double i,  
           @:integer|double j)  
    @:integer|double {  
      i + j  
    }
```

```
> annotations(adder, "body")  
[[1]]  
integer | double
```

# Proof-of-concept: contractR

```
a_fun <- function(@:any v) @:any { v }
i_fun <- function(@:integer v) @:integer { v+1L }
n_fun <- function(@:numeric v) @:numeric { v/2L }
nv_fun <- function(@:numeric[] v) @:numeric[] { v/2L }
lv_fun <- function(@:logical[2,] v) @:logical[2,] { !v }
cv_fun <- function(@:character[] v) @:character[] {paste0("!", v)}
```

# Proof-of-concept: contractR implementation

```
.onAttach <- function(libname, pkgname) {  
  register_annotation_handler(  
    "formals",  
    create_handler("arg", match_datatype, add_arg_contract))  
  
  register_annotation_handler(  
    "body",  
    create_handler("ret", match_datatype, add_ret_contract))  
}
```

# Proof-of-concept: contractR implementation

```
match_atomic <- function(t) {  
  switch(as.character(t),  
         logical = list(contract = is.logical, expected = t),  
         ... , FALSE)  
}  
  
match_dimensions <- function(dimensions) { ... }  
  
match_array <- function(t) { ... }  
  
match_datatype <- function(t) { if (is.symbol(t)) match_atomic(t)  
                                else match_array(t) }
```

# Proof-of-concept: contractR implementation

```
insert_arg_contract <- function(match, funname, fun, formal) {  
  match$funbody <- delimit_exprs(body(fun))  
  match$formal <- formal  
  match$funname <- funname  
  body(fun) <- substitute({  
    msg <- contractr:::argument_message(funname,  
                                         formal,  
                                         expected)  
    contractr:::failwith((contract)(formal), msg)  
    funbody  
  },  
  match)  
  fun  
}
```

# To see it in action

```
$ git clone https://github.com/aviralg/r-3.4.0.git
$ cd r-3.4.0
$ git checkout annotation
$ ./configure --with-recommended-packages
$ make -j
$ bin/R

> install.packages("devtools")
> library(devtools)
> devtools::install_github("aviralg/annotatr")
> devtools::install_github("aviralg/contractr")
> library(contractr)
```



# Why not a library?

With annotations

```
cSqrt <- function(@: +numeric x) sqrt(x)
```

```
cSqrt <- function(x) sqrt(x)
```

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

With typeinfo

# Why not a library?

With annotations

```
cSqrt <- function(@:+numeric x) sqrt(x)
```

```
cSqrt <- function(x) sqrt(x)
```

```
cSqrt <- function(x) {  
  assertNumeric(x, lower = 0)  
  sqrt(x)  
}
```

With checkmate

# Why not a library?

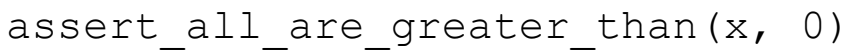
**With annotations**

`cSqrt <- function(@:+numeric x) sqrt(x)`



`cSqrt <- function(x) sqrt(x)`

`cSqrt <- function(x) {  
 assert_all_are_greater_than(x, 0)  
 sqrt(x)  
}`



**With assertive**

# Related Work

Annotations

Java, Scala, Kotlin, Groovy

Attributes

C#, F#, Swift

Decorators

Python, Javascript

# Related Work - Annotations

Annotations in **Java** can be applied to declarations and use of types.

```
@Author(name = "John Gee",  
        date = "7/04/2017")  
class Logger() { ... }  
  
name = (@NonNull String) entry.name;  
  
@SuppressWarnings("unchecked")  
void findName() { ... }
```

Annotations in **Scala** attach metadata with definitions.

```
object AnnotationDemo extends App {  
  @deprecated  
  def addtwo(x: Int): Int = x + 2  
  
  addtwo  
}
```

# Related Work - Attributes

Attributes In **Swift** provide information about declaration and types.

```
@available(macOS 10.12, iOS 2.0, *)
class Animal {
    // class definition
}
```

This specifies platform availability of Animal class.

Attributes In **C#** are either predefined or user-defined custom information attached to language elements that can be examined at run-time.

```
[Obsolete("Please use CreatePost")]
public Post NewPost()
{

}
```

# Related Work - Decorators

Decorators in **Python** are functions that take a function or method as an argument and return a callable. Their execution semantics is baked into the language.

```
@dec1  
@dec2  
@dec3  
def my_function(arg) ...
```

is equivalent to

```
my_function = dec1(dec2(dec3(my_function)))
```

# Annotations ...

... modify syntax to attach arbitrary metadata to language objects

... are available through reflection

... do not impose any execution semantics

... are optional

... do not affect existing code



**@:thanks**