A Method for Deriving Information from Running R Code

by Mark P.J. van der Loo

Abstract It is often useful to tap information from a running R script. Obvious use cases include monitoring the consumption of resources (time, memory) and logging. Perhaps less obvious cases include tracking changes in R objects or collecting the output of unit tests. In this paper, we demonstrate an approach that abstracts the collection and processing of such secondary information from the running R script. Our approach is based on a combination of three elements. The first element is to build a customized way to evaluate code. The second is labeled *local masking* and it involves temporarily masking a user-facing function so an alternative version of it is called. The third element we label *local side effect*. This refers to the fact that the masking function exports information to the secondary information flow without altering a global state. The result is a method for building systems in pure R that lets users create and control secondary flows of information with minimal impact on their workflow and no global side effects.

1 Introduction

The R language provides a convenient language to read, manipulate, and write data in the form of scripts. As with any other scripted language, an R script gives a description of data manipulation activities, one after the other, when read from top to bottom. Alternatively, we can think of an R script as a one-dimensional visualization of data flowing from one processing step to the next, where intermediate variables or pipe operators carry data from one treatment to the next.

We run into limitations of this one-dimensional view when we want to produce data flows that are somehow 'orthogonal' to the flow of the data being treated. For example, we may wish to follow the state of a variable while a script is being executed, report on progress (logging), or keep track of resource consumption. Indeed, the sequential (one-dimensional) nature of a script forces one to introduce extra expressions between the data processing code.

As an example, consider a code fragment where the variable x is manipulated.

```
x[x > threshold] <- threshold
x[is.na(x)]  <- median(x, na.rm=TRUE)
```

In the first statement, every value above a certain threshold is replaced with a fixed value, and next, missing values are replaced with the median of the completed cases. It is interesting to know how an aggregate of interest, say the mean of x, evolves as it gets processed. The instinctive way to do this is to edit the code by adding statements to the script that collect the desired information.

```
meanx <- mean(x, na.rm=TRUE)
x[x > threshold] <- threshold
meanx <- c(meanx, mean(x, na.rm=TRUE))
x[is.na(x)]  <- median(x, na.rm=TRUE)
meanx <- c(meanx, mean(x, na.rm=TRUE))
```

This solution clutters the script by inserting expressions that are not necessary for its main purpose. Moreover, the tracking statements are repetitive, which validates some form of abstraction.

A more general picture of what we would like to achieve is given in Figure 1. The 'primary data flow' is developed by a user as a script. In the previous example, this concerns processing x. When the script runs, some kind of logging information, which we label the 'secondary data flow' is derived implicitly by an abstraction layer.

Creating an abstraction layer means that concerns between primary and secondary data flows are separated as much as possible. In particular, we want to prevent the abstraction layer from inspecting or altering the user code that describes the primary data flow. Furthermore, we would like the user to have some control over the secondary flow from within the script, for example, to start, stop, or parameterize the secondary flow. This should be done with minimum editing of the original user code, and it should not rely on global side effects. This means that neither the user nor the abstraction layer for the secondary data flow should have to manipulate or read global variables, options, or other environmental settings to convey information from one flow to the other. Finally, we want to treat the availability of a secondary data flow as a normal situation. This means we wish to avoid using signaling conditions (e.g., warnings or errors) to convey information between the flows unless there is an actual exceptional condition such as an error.

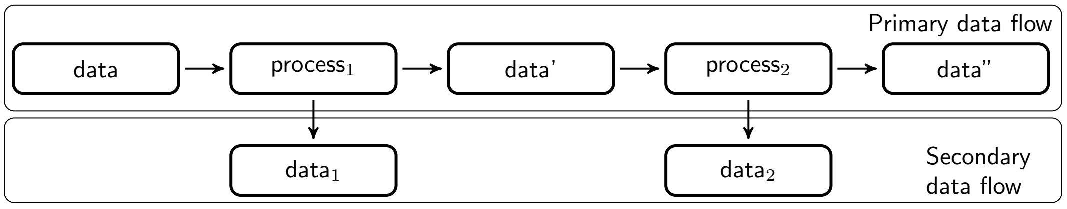

Figure 1: Primary and secondary data flows in an R script. The primary flow follows the execution of an R script, while in the background a secondary data flow (e.g. logging information) is created.

Prior art

There are several packages that generate a secondary data flow from a running script. One straightforward application concerns logging messages that report on the status of a running script. To create a logging message, users edit their code by inserting logging expressions where desired. Logging expressions are functions calls that help to build expressions, for example, by automatically adding a timestamp. Configuration options usually include a measure of logging verbosity and setting an output channel that controls where logging data will be sent. Changing these settings relies on communication from the main script to the functionality that controls the flow of logging data. In **logger** (Daróczi, 2021), this is done by manipulating a variable stored in the package namespace using special helper functions. The **logging** package (Frasca, 2019) also uses an environment within the namespace of the package to manage option settings, while **futile.logger** (Rowe, 2016) implements a custom global option settings manager that is somewhat comparable to R's own `options()` function.

Packages **bench** (Hester, 2020b) and **microbenchmark** (Mersmann, 2019) provide time profiling of single R expressions. The **bench** package also includes memory profiling. Their purpose is not to derive a secondary data flow from a running production script as in Figure 1 but to compare the performance of R expressions. Both packages export a function that accepts a sequence of expressions to profile. These functions take control of expression execution and insert time and/or memory measurements where necessary. Options, such as the number of times each expression is executed, are passed directly to the respective function.

Unit testing frameworks provide another source of secondary data flows. Here, an R script is used to prepare, set up, and compare test data, while the results of comparisons are tapped and reported. Testing frameworks are provided by **testthat** (Wickham, 2011), **RUnit**, (Burger et al., 2018), **testit** Xie (2021), **unitizer** (Gaslam, 2021), and **tinytest** (van der Loo, 2020). The first three packages (**testthat**, **RUnit**, and **testit**) all export assertion functions that generate condition signals to convey information about test results. Packages **RUnit** and **testit** use `sys.source()` to run a file containing unit test assertions and exit on the first error while **testthat** uses `eval()` to run expressions, capture conditions, and test results and reports afterward. The **unitizer** framework is different because it implements an interactive prompt to run tests and explore their results. Rather than providing explicit assertions, **unitizer** stores results of all expressions that return a visible result and compares their output at subsequent runs. Interestingly, **unitizer** allows for optional monitoring of the testing environment. This includes environment variables, options, and more. This is done by manipulating the code of (base) R functions that manage these settings and masking the original functions temporarily. These masking functions then provide parts of the secondary data flow (changes in the environment). Finally, **tinytest** is based on the approach that is the topic of this paper, and it will be discussed as an application below.

Finally, we note the **covr** package of Hester (2020a). This package is used to keep track of which expressions of an R package are run (covered) by package tests or examples. In this case, the primary data flow is a test script executing code (functions, methods) stored in another script, usually in the context of a package. The secondary flow consists of counts of how often each expression in the source is executed. The package works by parsing and altering the code in the source file, inserting expressions that increase appropriate counters. These counters are stored in a variable that is part of the package's namespace.

Summarizing, we find that in logging packages, the secondary data flow is invoked explicitly by users while configuration settings are communicated by manipulating a global state that may or may not be directly accessible by the user. For benchmarking packages, the expressions are passed explicitly to an 'expression runner' that monitors the effect on memory and passage of time. In most test packages, the secondary flow is invoked explicitly using special assertions that throw condition signals. Test files are run using functionality that captures and administrates signals where necessary. Two of the discussed packages explicitly manipulate existing code before running it to create a secondary data flow. The **covr** package does this to update expression counters and the

unitizer package to monitor changes in the global state.

Contribution of this paper

The purpose of this paper is to first provide some insight into the problem of managing multiple data flows, independent of specific applications. In the following section, we discuss managing a secondary data stream from the point of view of changing the way in which expressions are combined and executed by R.

Next, we highlight two programming patterns that allow one to derive a secondary data stream, both in non-interactive (while executing a file) and in interactive circumstances. The methods discussed here do not require explicit inspection or modification of the code that describes the primary data flow. It is also not necessary to invoke signaling conditions to transport information from or to the secondary data stream.

We also demonstrate a combination of techniques that allow users to parameterize the secondary flow without resorting to global variables, global options, or variables within a package's namespace. We call this technique 'local masking' with 'local side effects'. It is based on temporarily and locally masking a user-facing function with a function that does exactly the same except for a side effect that passes information to the secondary data flow.

As examples, we discuss two applications where these techniques have been implemented. The first is the **lumberjack** package (van der Loo, 2021), which allows for tracking changes in R objects as they are manipulated expression by expression. The second is **tinytest** (van der Loo, 2020), a compact and extensible unit testing framework.

Finally, we discuss some advantages and limitations to the techniques proposed.

2 Concepts

In this section we give a high-level overview of the problem of adding a second data flow to an existing one, and general way to think about a solution. The general approach was inspired by a discussion of Milewski (2018) and is related to what is sometimes called a *bind operator* in functional programming.

Consider as an example the following two expressions, labeled e_1 and e_2.

```
e1:  x <- 10
e2:  y <- 2*x
```

We would like to implement some kind of monitoring as these expressions are evaluated. For this purpose, it is useful to think of e_1 and e_2 as functions that accept a set of key-value pairs, possibly alter the set's contents, and return it. In R this set of key-value pairs is an `environment`, and usually, it is the global environment (the user's workspace). Starting with an empty environment $\{\}$ we get:

$$e_1(\{\}) = \{(\texttt{"x"},\texttt{10})\}$$
$$e_2(e_1(\{\})) = \{(\texttt{"x"},\texttt{10}),(\texttt{"y"},\texttt{20})\}$$

In this representation, we can write the result of executing the above script in terms of the function composition operator $\circ$:

$$e_2(e_1(\{\})) = (e_2 \circ e_1)(\{\}).$$

And in general, we can express the final state $\mathcal{U}$ of any environment after executing a sequence of expressions $e_1, e_2, \cdots, e_k$ as:

$$\mathcal{U} = (e_k \circ e_{k-1} \circ \cdots \circ e_1)(\{\}), \tag{1}$$

where we assumed without loss of generality that we start with an empty environment. We will refer to the sequence $e_1 \ldots e_k$ as the 'primary expressions' since they define a user's main data flow.

We now wish to introduce some kind of logging. For example, we want to count the number of evaluated expressions, not counting the expressions that will perform the count. The naive way to do this is to introduce a new expression, say n:

```
n:   if (!exists("N")) N <- 1 else N <- N + 1
```

And we insert this into the original sequence of expressions. This amounts to the cumbersome solution:

$$\mathcal{U} \cup \{(\texttt{"N"},k)\} = (n \circ e_k \circ n \circ e_{k-1} \circ n \circ \cdots n \circ e_1)(\{\}), \tag{2}$$

where the number of executed expressions is stored in `N`. We shall refer to `n` as a 'secondary expression', as it does not contribute to the user's primary data flow.

The above procedure can be simplified if we define a new function composition operator $\circ_n$ as follows:

$$a \circ_n b = a \circ n \circ b.$$

One may verify the associativity property $a \circ_n (b \circ_n c) = (a \circ_n b) \circ_n c$ for expressions a, b, and c, so $\circ_n$ can, indeed, be interpreted as a new function composition operator. Using this operator we get

$$\mathcal{U} \cup \{(\texttt{"N"}, k-1)\} = (e_k \circ_n e_{k-1} \circ_n \cdots \circ_n e_1)(\{\}), \tag{3}$$

which gives the same result as Equation 2 up to a constant.

If we are able to alter function composition, then this mechanism can be used to track all sorts of useful information during the execution of $e_1, \ldots, e_k$. For example, a simple profiler is set up by timing the expressions and adding the following expression to the function composition operator.

```
s: if (!exists("S")) S <- Sys.time() else S <- c(S, Sys.time())
```

After running $e_k \circ_s \cdots \circ_s e_1$, `diff(S)` gives the timings of individual statements. A simple memory profiler is defined as follows.

```
m: if (!exists("M")) M <- sum(memory.profile()) else M <- c(M, sum(memory.profile()))
```

After running $e_k \circ_m \cdots \circ_m e_1$, `M` gives the amount of memory used by R after each expression.

We can also track changes in data, but it requires that the composition operator knows the name of the R object that is being tracked. As an example, consider the following primary expressions.

```
e1:  x <- rnorm(10)
e2:  x[x<0] <- 0
e3:  print(x)
```

We can define the following expression for our modified function composition operator.

```
v:  {
      if (!exists("V")){
        V <- logical(0)
        x0 <- x
      }
      if (identical(x0,x)) V <- c(V, FALSE)
      else V <- c(V, TRUE)
      x0 <- x
    }
```

After running $e_3 \circ_v e_2 \circ_v e_1$, the variable `V` equals `c(TRUE,FALSE)`, indicating that e_2 changed `x`, and e_3 did not.

These examples demonstrate that redefining function composition yields a powerful method for extracting logging information with (almost) no intrusion on the user's common workflow. The simple model shown here does have some obvious setbacks: first, the expressions inserted by the composition operator manipulate the same environment as the user expressions. The user- and secondary expressions can therefore interfere with each other's results. Second, there is no direct control from the primary sequence over the secondary sequence: the user has no explicit control over starting, stopping, or parametrizing the secondary data stream. We demonstrate in the next section how these setbacks can be avoided by evaluating secondary expressions in a separate environment and by using techniques we call 'local masking' and 'local side-effect'.

3 Creating a secondary data flow with R

R executes expressions one by one in a read-evaluate-print loop (REPL). In order to tap information from this running loop, it is necessary to catch the user's expressions and interweave them with our own expressions. One way to do this is to develop an alternative to R's native `source()` function. Recall that `source()` reads an R script and executes all expressions in the global environment. Applications include non-interactive sessions or interactive sessions with repetitive tasks such as running test scripts while developing functions. A second way to intervene with a user's code is to develop a special 'forward pipe' operator, akin to R's `|>` pipe, the **magrittr** pipe of Bache and Wickham (2014), or the 'dot-pipe' of Mount and Zumel (2018). Since a user inserts a pipe between expressions, it is an obvious place to insert code that generates a secondary data flow.

In the following two subsections we will develop both approaches. As a running example, we will implement a secondary data stream that counts expressions.

Build your own `source()`

The `source()` function reads an R script and executes all expressions in the global environment. A simple variant of `source()` that counts expressions as they get evaluated can be built using `parse()` and `eval()`.

```
run <- function(file){
  expressions <- parse(file)
  runtime <- new.env(parent=.GlobalEnv)

  n <- 0
  for (e in expressions){
    eval(e, envir=runtime)
    n <- n + 1
  }
  message(sprintf("Counted %d expressions",n))
  runtime
}
```

Here, `parse()` reads the R file and returns a list of expressions (technically, an object of class '`expression`'). The `eval()` function executes the expression while all variables created by or needed for execution are sought in a newly created environment called `runtime`. We make sure that variables and functions in the global environment are found by setting the parent of `runtime` equal to `.GlobalEnv`. Now, given a file `"script.R"`.

```
# contents of script.R
x <- 10
y <- 2*x
```

An interactive session would look like this.

```
> e <- run("script.R")
Counted 2 expressions
> e$x
[1] 10
```

So, contrary to the default behavior of `source()`, variables are assigned in a new environment. This difference in behavior can be avoided by evaluating expressions in `.GlobalEnv`. However, for the next step, it is important to have a separate runtime environment.

We now wish to give the user some control over the secondary data stream. In particular, we want the user to be able to choose when `run()` starts counting expressions. Recall that we demand that this is done by direct communication to `run()`. This means that side-effects such as setting a special variable in the global environment or a global option is out of the question. Furthermore, we want to avoid code inspection: the `run()` function should be unaware of what expressions it is running exactly. We start by writing a function for the user that returns `TRUE`.

```
start_counting <- function() TRUE
```

Our task is to capture this output from `run()` when `start_counting()` is called. We do this by masking this function with another function that does exactly the same, except that it also copies the output value to a place where `run()` can find it. To achieve this, we use the following helper function.

```
capture <- function(fun, envir){
  function(...){
    out <- fun(...)
    envir$counting <- out
    out
  }
}
```

This function accepts a function (`fun`) and an environment (`envir`). It returns a function that first executes `fun(...)`, copies its output value to `envir`, and then returns the output to the user. In an interactive session, we would see the following.

```
> store <- new.env()
> f <- capture(start_counting, store)
> f()
[1] TRUE
> store$counting
[1] TRUE
```

Observe that our call to `f()` returns `TRUE` as expected but also exported a copy of `TRUE` into `store`. The reason this works is that an R function 'remembers' where it is created. The function `f()` was created inside `capture()`, and the variable `envir` is present there. We say that this 'capturing' version of `start_counting` has a *local side-effect*: it writes outside of its own scope but the place where it writes is controlled.

We now need to make sure that `run()` executes the captured version of `start_counting()`. This is done by locally masking the user-facing version of `start_counting()`. That is, we make sure that the captured version is found by `eval()` and not the original version. A new version of `run()` now looks as follows.

```
run <- function(file){
  expressions <- parse(file)
  store <- new.env()
  runtime <- new.env(parent=.GlobalEnv)
  runtime$start_counting <- capture(start_counting, store)
  n <- 0
  for (e in expressions){
    eval(e, envir=runtime)
    if ( isTRUE(store$counting) ) n <- n + 1
  }
  message(sprintf("Counted %d expressions",n))
  runtime
}
```

Now, consider the following code, stored in `script1.R`.

```
# contents of script1.R
x <- 10
start_counting()
y <- 2*x
```

In an interactive session, we would see this.

```
> e <- run("script1.R")
Counted 1 expressions
> e$x
[1] 10
> e$y
[1] 20
```

Let us go through the most important parts of the new `run()` function. After parsing the R file, a new environment is created that will store the output of calls to `start_counting()`.

```
  store  <- new.env()
```

The runtime environment is created as before, but now we add the capturing version of `start_counting()`.

```
  runtime <- new.env(parent=.GlobalEnv)
  runtime$start_counting <- capture(start_counting, store)
```

This ensures that when the user calls `start_counting()`, the capturing version is executed. We call this technique *local masking* since the `start_counting()` function is only masked during the execution of `run()`. The captured version of `start_counting()`as a side effect stores its output in `store`. We call this a 'local side-effect' because `store` is never seen by the user: it is created inside `run()` and destroyed when `run()` is finished.

Finally, all expressions are executed in the runtime environment and counted conditional on the value of `store$counting`.

```
  for (e in expressions){
    eval(e, envir=runtime)
    if ( isTRUE(store$counting) ) n <- n + 1
  }
```

Summarizing, with this construction, we are able to create a file runner akin to `source()` that can gather and communicate useful process metadata while executing a script. Moreover, the user of the script can convey information directly to the file runner, while it runs, without relying on global side-effects. This is achieved by first creating a user-facing function that returns the information to be sent to the file runner. The file runner locally masks the user-facing version with a version that copies the output to an environment local to the file runner before returning the output to the user.

The approach just described can be generalized to more realistic use cases. All examples mentioned in the 'Context' section —time or memory profiling, or logging changes in data, merely need some extra administration. Furthermore, the current example emits the secondary data flow as a '`message`'. In practical use cases, it may make more sense to write the output to a file connection or database or the make the secondary data stream output of the file runner. In the Applications section, both applications are discussed.

Build your own pipe operator

Pipe operators have become a popular tool for R users over the last years, and R currently has a pipe operator (`|>`) built-in. This pipe operator is intended as a form of 'syntactic sugar' that, in some cases, makes code a little easier to write. A pipe operator behaves somewhat like a left-to-right 'expression composition operator'. This, in the sense that a sequence of expressions that are joined by a pipe operator are interpreted by R's parser as a single expression. Pipe operators also offer an opportunity to derive information from a running sequence of expressions.

It is possible to implement a basic pipe operator as follows.

```
`%p>%` <- function(lhs, rhs) rhs(lhs)
```

Here, the `rhs` (right-hand side) argument must be a single-argument function, which is applied to `lhs`. In an interactive session we could see this.

```
> 3 %p>% sin %p>% cos
[1] 0.9900591
```

To build our expression counter, we need to have a place to store the counter value hidden from the user. In contrast to the implementation of the file runner in the previous section, each use of `%p>%` is disconnected from the other, and there seems to be no shared space to increase the counter at each call. The solution is to let the secondary data flow travel with the primary flow by adding an attribute to the data. We create two user-facing functions that start or stop logging as follows.

```
start_counting <- function(data){
  attr(data, "n") <- 0
  data
}
end_counting <- function(data){
  message(sprintf("Counted %d expressions", attr(data,"n")-1))
  attr(data, "n") <- NULL
  data
}
```

Here, the first function attaches a counter to the data and initializes it to zero. The second function reports its value, decreased by one, so the stop function itself is not included in the count. We also alter the pipe operator to increase the counter if it exists.

```
`%p>%` <- function(lhs, rhs){
  if ( !is.null(attr(lhs,"n")) ){
    attr(lhs,"n") <- attr(lhs,"n") + 1
  }
  rhs(lhs)
}
```

In an interactive session, we could now see the following.

```
> out <- 3 %p>%
+  start_counting %p>%
+    sin %p>%
+    cos %p>%
+  end_counting
Counted 2 expressions
```

```
> out
[1] 0.9900591
```

Summarizing, for small interactive tasks, a secondary data flow can be added to the primary one by using a special kind of pipe operator. Communication between the user and the secondary data flow is implemented by adding or altering attributes attached to the R object.

Generalizations of this technique come with a few caveats. First, the current pipe operator only allows right-hand side expressions that accept a single argument. Extension to a more general case involves inspection and manipulation of the right-hand side's abstract syntax tree and is out of scope for the current work. Second, the current implementation relies on the right-hand side expressions to preserve attributes. A general implementation will have to test that the output of rhs(lhs) still has the logging attribute attached (if there was any) and re-attach it if necessary.

4 Application 1: tracking changes in data

The **lumberjack** package (van der Loo, 2021) implements a logging framework to track changes in R objects as they get processed. The package implements both a pipe operator, denoted `%L>%`, and a file runner called `run_file()`. The main communication devices for the user are two functions called `start_log()` and `dump_log()`.

We will first demonstrate working with the **lumberjack** pipe operator. The function `start_log()` accepts an R object and a logger object. It attaches the logger to the R object and returns the augmented R object. A logger is a reference object[1] that exposes at least an `$add()` method and a `$dump()` method. If a logger is present, the pipe operator stores a copy of the left-hand side. Next, it executes the expression on the right-hand side with the left-hand side as an argument and stores the output. It then calls the `add()` method of the logger with the input and output so that the logger can compute and store the difference. The `dump_log()` function accepts an R object, calls the `$dump()` method on the attached logger (if there is any), removes the logger from the object and returns the object. An interactive session could look as follows.

```
> library(lumberjack)
> out <- women %L>%
>   start_log(simple$new()) %L>%
>   transform(height = height * 2.54) %L>%
>   identity() %L>%
>   dump_log()
Dumped a log at /home/mark/simple.csv
> read.csv("simple.csv")
  step                time                        expression changed
1    1 2019-08-09 11:29:06 transform(height = height * 2.54)    TRUE
2    2 2019-08-09 11:29:06                        identity()   FALSE
```

Here, `simple$new()` creates a logger object that registers whether an R object has changed or not. There are other loggers that compute more involved differences between in- and output. The `$dump()` method of the logger writes the logging output to a csv file.

For larger scripts, a file runner called `run_file()` is available in **lumberjack**. As an example, consider the following script. It converts columns of the built-in `women` data set to SI units (meters and kilogram) and then computes the body-mass index of each case.

```
# contents of script2.R
start_log(women, simple$new())
women$height <- women$height * 2.54/100
women$weight <- women$weight * 0.453592
women$bmi    <- women$weight/(women$height)^2
```

In an interactive session, we can run the script and access both the logging information and retrieve the output of the script.

```
> e <- run_file("script2.R")
Dumped a log at /home/mark/women_simple.csv
> read.csv("women_simple.csv")
  step                time                            expression changed
1    1 2019-08-09 13:11:25         start_log(women, simple$new())   FALSE
```

[1]A native R Reference Class, an 'R6' object (Chang, 2020), or any other reference type object implementing the proper API.

```
2    2 2019-08-09 13:11:25    women$height <- women$height * 2.54/100    TRUE
3    3 2019-08-09 13:11:25    women$weight <- women$weight * 0.453592    TRUE
4    4 2019-08-09 13:11:25 women$bmi <- women$weight/(women$height)^2    TRUE
> head(e$women,3)
  height   weight      bmi
1 1.4732 52.16308 24.03476
2 1.4986 53.07026 23.63087
3 1.5240 54.43104 23.43563
```

The **lumberjack** file runner locally masks `start_log()` with a function that stores the logger and the name of the tracked R object in a local environment. A copy of the tracked object is stored locally as well. Expressions in the script are executed one by one. After each expression, the object in the runtime environment is compared with the stored object. If it has changed, the `$add()` method of the logger is called, and a copy of the changed object is stored. After all expressions have been executed, the `$dump()` method is called, so the user does not have to do this explicitly.

A user can add multiple loggers for each R object and track multiple objects. It is also possible to dump specific logs for specific objects during the script. All communication necessary for these operations runs via the mechanism explained in the 'build your own `source()`' section.

5 Application 2: unit testing

The **tinytest** package (van der Loo, 2020) implements a unit testing framework. Its core function is a file runner that uses local masking and local side effects to capture the output of assertions that are inserted explicitly by the user. As an example, we create tests for the following function.

```
# contents of bmi.R
bmi <- function(weight, height) weight/(height^2)
```

A simple **tinytest** test file could look like this.

```
# contents of test_script.R
data(women)
women$height <- women$height * 2.54/100
women$weight <- women$weight * 0.453592
BMI    <- with(women, bmi(weight,height) )

expect_true( all(BMI >= 10) )
expect_true( all(BMI <= 30) )
```

The first four lines prepare some data, while the last two lines check whether the prepared data meets our expectations. In an interactive session, we can run the test file after loading the `bmi()` function.

```
> source("bmi.R")
> library(tinytest)
> out <- run_test_file('test_script.R')
Running test_script.R................    2 tests OK
> print(out, passes=TRUE)
----- PASSED      : test_script.R<7--7>
 call| expect_true(all(BMI >= 10))
----- PASSED      : test_script.R<8--8>
 call| expect_true(all(BMI <= 30))
```

In this application, the file runner locally masks the `expect_*()` functions and captures their result through a local side effect. As we are only interested in the test results, the output of all other expressions is discarded.

Compared to the basic version described in the 'build your own `source()`' section, this file runner keeps some extra administration, such as the line numbers of each masked expression. These can be extracted from the output of `parse()`. The package comes with a number of assertions in the form of `expect_*()` functions. It is possible to extend **tinytest** by registering new assertions. These are then automatically masked by the file runner. The only requirement on the new assertions is that they return an object of the same type as the built-in assertions (an object of class `'tinytest'`).

6 Discussion

The techniques demonstrated here have two major advantages. First, it allows for a clean and side-effect free separation between the primary and secondary data flows. As a result, the secondary data flow is composed with the primary data flow. In other words: a user that wants to add a secondary data flow to an existing script does not have to edit any existing code. Instead, it is only necessary to add a bit of code to specify and initialize the secondary stream, which is a big advantage for maintainability. Second, the current mechanisms avoid the use of condition signals. This also leads to code that is easier to understand and navigate because all code associated with the secondary flow can be limited to the scope of a single function (here: either a file runner or a pipe operator). Since the secondary data flow is not treated as an unusual condition (exception), the exception signaling channel is free for transmitting truly unusual conditions such as errors and warnings.

There are also some limitations inherent to these techniques. Although the code for the secondary data flow is easy to compose with code for the primary data flow, it is not as easy to compose different secondary data flows. For example, one can use only one file runner to run an R script and only a single pipe operator to combine two expressions.

A second limitation is that this approach does not recurse into the primary expressions. For example, the expression counters we developed only count user-defined expressions. They can not count expressions that are called by functions called by the user. This means that something like a code coverage tool such as **covr** is out of scope.

A third and related limitation is that the resolution of expressions may be too low for certain applications. For example in R, '`if`' is an expression (it returns a value when evaluated) rather than a statement (like `for`). This means that `parse()` interprets a block such as

```
if ( x > 0 ){
 x <- 10
 y <- 2*x
}
```

as a single expression. If higher resolution is needed, this requires explicit manipulation of the user code.

Finally, the local masking mechanism excludes the use of the namespace resolution operator. For example, in **lumberjack**, it is not possible to use `lumberjack::start_log()` since, in that case, the user-facing function from the package is executed and not the masked function with the desired local side-effect.

7 Conclusion

In this paper we demonstrated a set of techniques that allow one to add a secondary data flow to an existing user-defined R script. The core idea is that we manipulate way expressions are combined before they are executed. In practice, we use R's `parse()` and `eval()` to add secondary data stream to user code, or build a special 'pipe' operator. Local masking and local side effects allow a user to control the secondary data flow without global side-effects. The result is a clean separation of concerns between the primary and secondary data flow, that does not rely on condition handling, is void of global side-effects, and that is implemented in pure R.

Mark P.J. van der Loo
Statistics Netherlands
PO-BOX 24500, 2490HA Den Haag
The Netherlands
https://orcid.org/0000-0002-9807-4686
https://www.markvanderloo.eu
m.vanderloo@cbs.nl

Bibliography

S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL https://CRAN.R-project.org/package=magrittr. R package version 1.5. [p45]

M. Burger, K. Juenemann, and T. Koenig. *RUnit: R Unit Test Framework*, 2018. URL https://CRAN.R-project.org/package=RUnit. R package version 0.4.32. [p43]

W. Chang. *R6: Encapsulated Classes with Reference Semantics*, 2020. URL `https://CRAN.R-project.org/package=R6`. R package version 2.5.0. [p49]

G. Daróczi. *logger: A Lightweight, Modern and Flexible Logging Utility*, 2021. URL `https://CRAN.R-project.org/package=logger`. R package version 0.2.0. [p43]

M. Frasca. *logging: R Logging Package*, 2019. URL `https://CRAN.R-project.org/package=logging`. R package version 0.10-108. [p43]

B. Gaslam. *unitizer: Interactive R Unit Tests*, 2021. URL `https://CRAN.R-project.org/package=unitizer`. R package version 1.4.14. [p43]

J. Hester. *covr: Test Coverage for Packages*, 2020a. URL `https://CRAN.R-project.org/package=covr`. R package version 3.5.1. [p43]

J. Hester. *bench: High Precision Timing of R Expressions*, 2020b. URL `https://CRAN.R-project.org/package=bench`. R package version 1.1.1. [p43]

O. Mersmann. *microbenchmark: Accurate Timing Functions*, 2019. URL `https://CRAN.R-project.org/package=microbenchmark`. R package version 1.4-7. [p43]

B. Milewski. *Category Theory for Programmers*. Blurb, Incorporated, 2018. ISBN 9780464825081. See also the online lectures: https://youtu.be/I8LbkfSSR58. [p44]

J. Mount and N. Zumel. Dot-Pipe: an S3 Extensible Pipe for R. *The R Journal*, 10(2):309–316, 2018. doi: 10.32614/RJ-2018-042. URL `https://doi.org/10.32614/RJ-2018-042`. [p46]

B. L. Y. Rowe. *futile.logger: A Logging Utility for R*, 2016. URL `https://CRAN.R-project.org/package=futile.logger`. R package version 1.4.3. [p43]

M. van der Loo. *tinytest: Lightweight but Feature Complete Unit Testing Framework*, 2020. URL `https://github.com/markvanderloo/tinytest`. R package version 1.2.4. [p43, 44, 50]

M. P. J. van der Loo. Monitoring Data in R with the lumberjack Package. *Journal of Statistical Software*, 98:1–13, 2021. [p44, 49]

H. Wickham. testthat: Get started with testing. *The R Journal*, 3:5–10, 2011. URL `https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf`. [p43]

Y. Xie. *testit: A Simple Package for Testing R Packages*, 2021. URL `https://CRAN.R-project.org/package=testit`. R package version 0.13. [p43]