Reactive Programming with Algebra

André van Delft
Independent Researcher
andre dot vandelft at gmail dot com

Anatoliy Kmetyuk
Student, Odessa National Academy of Law
anatoliykmetyuk at gmail dot com

Abstract

R&D on reactive programming is growing and this has delivered many language constructs, libraries and tools. Scala programmers can use threads, timers, actors, futures, promises, observables, the async construct, and others. Still it seems to us that the state of the art is not mature: reactive programming is relatively hard, and confidence in correct operation depends mainly on extensive testing. Better support for reasoning about correctness would be useful to address timing dependent issues.

The Algebra of Communicating Processes (ACP) has a potential to improve this: it lets one concisely specify event handling and concurrency, and it facilitates reasoning about parallel systems. There is an ACP-based extension to Scala named SubScript, which also adds language level support for data flow and actor message handling. We show how SubScript helps specifying the control flow of reactive programs.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Dataflow languages

General Terms Languages, Theory

Keywords Algebra of Communicating Processes, ACP, data flow, concurrency, non-determinism, reactive programming, actors, futures, Akka

1. Introduction

As argued in the Reactive Manifesto[1], the need is growing for applications that are event-driven, scalable, resilient and responsive. Reactive programming gains attention, particularly in the Scala world. Developers can use the Akka toolkit[2], which offers abstractions such as actors [7], futures [4] and state transition machines (STMs). Akka has become a solid programming platform. Its message throughput suits many applications: in the order of magnitude of one million per second, on a modern CPU.

Yet we think that the way internal behavior of Akka actors is expressed, should be improved. Actors need to process incoming messages; send messages on to other actors and be ready for their responses; have timeouts for such expectations; and all this concurrently. It is possible to express such tasks in both object-oriented and functional languages, and certainly in Scala. Yet we think that the current Akka programs are a bit obscure:

- Actors may have a *receive* method for processing incoming messages; state is then usually maintained using instance variables. The receive method is essentially a list of incoming message kinds and the associated handlers. The typical order for such messages has no clear relation with the program text.
- Alternatively actors may contain an STM: a `become` method allows to change to a parameterized state. This overcomes part of the unclarity of the *receive* method, but the code may seem a bit spaghetti-like. `become` is a kind of jump, a higher level reminiscent of the goto instruction in the doghouse.
- Futures and async constructs are available for concurrency and time management inside a single actor. Futures are high level abstractions for convenient expressing data dependencies while allowing for concurrency; however in our opinion these constructs obfuscate the control flow somewhat.

For us this was illustrated during the Coursera course "Principles on Reactive Programming" that we both took in November and December 2013[3]. Going from the natural language specification to a working program seemed very hard. We did not feel well about our solutions to the main actor programming assignment: these passed both our tests and the tests created by the course leaders, but the program text was not clear enough to reason easily about correctness, e.g. with respect to time dependencies and race conditions.

One of us was helped by designing the actor behavior for

[1] http://www.reactivemanifesto.org/

[2] http://akka.io/

[3] https://www.coursera.org/course/reactive

this assignment in the formalism of the Algebra of Communicating Processes (ACP). This is a concurrency theory that allows for concise specifications of event-driven and concurrent processes. It also helps formal reasoning about process behavior. ACP is good at describing processes that communicate synchronously, and less at describing asynchronously communicating processes, such as actors. Thus it seems well suited for describing internal actor behavior, and, for the time being at least, not for complete actor systems.
It is also possible to program applications using ACP. We are developing an ACP based extension to Scala by the name of SubScript. This paper is a follow up to a paper presented at the Scala Workshop 2013 [11] about dataflow programming support in SubScript, with applications to actor systems. We present in this paper the ideas that have evolved since then, with applications to dataflow and actor programming. Futures and ACP processes have some common properties. Apparently they may be mixed in SubScript programs, by virtue of the Scala's implicit features.
A SubScript implementation is available. It comes with a compiler, which is a derivative of the regular Scala compiler. This transforms ACP-like specifications into calls to an API of a SubScript Virtual Machine. There are also compatibility layer, for the Swing and Akka frameworks.
The rest of this paper is structured as follows: we introduce ACP, SubScript (first by two examples, then by features), the "call graph" semantics, and the basic implementation.[4] Section 7 and 8 highlight dataflow programming and actor programming with SubScript. The final sections discuss interoperation with futures, performance and some related work.

2. ACP

The Algebra of Communicating Processes [2] is an algebraic approach to reasoning about concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras or process calculi[5]. More so than the other seminal process calculi (CCS [9] and CSP [8]), the development of ACP focused on the algebra of processes, and sought to create an abstract, generalized axiomatic system for processes.
ACP uses instantaneous, atomic actions (a,b,c,...) as its main primitives. Two special primitives are the deadlock process δ and the empty process ϵ. Expressions of primitives and operators represent processes. The main operators can be roughly categorized as providing a basic process algebra, concurrency, and communication:

- *Choice and sequencing* - the most fundamental of algebraic operators are the alternative operator ($+$), which provides a choice between actions, and the sequencing operator ($\cdot$), which specifies an ordering on actions. So, for example, the process $(a+b)\cdot c$ first chooses to perform either a or b, and then performs action c. How the choice between a and b is made does not matter and is left unspecified. Note that alternative composition is commutative but sequential composition is not (because time flows forward).
- *Concurrency* - to allow the description of concurrency, ACP provides the merge operator $\|$. This represents the parallel composition of two processes, the individual actions of which are interleaved. As an example, the process $(a \cdot b) \parallel (c \cdot d)$ may perform the actions a, b, c, d in any of the sequences abcd, acbd, acdb, cabd, cadb, cdab.
- *Communication* - pairs of atomic actions may be defined as communicating actions, implying they can not be performed on their own, but only together, when active in two parallel processes. This way, the two processes synchronize, and they may exchange data.

ACP fundamentally adopts an axiomatic, algebraic approach to the formal definition of its various operators. Using the alternative and sequential composition operators, ACP defines a basic process algebra which satisfies the following axioms:

$$\begin{aligned} x + y &= y + x \\ (x + y) + z &= x + (y + z) \\ x + x &= x \\ (x + y) \cdot z &= x \cdot z + y \cdot z \\ (x \cdot y) \cdot z &= x \cdot (y \cdot z) \end{aligned}$$

The primitives 0 and 1, also known as δ and ϵ, behave much like the 0 and 1 that are usually neutral elements for addition and multiplication in algebra:

$$\begin{aligned} 0 + x &= x \\ 0 \cdot x &= \delta \\ 1 \cdot x &= x \\ x \cdot 1 &= x \end{aligned}$$

There is no axiom for $x \cdot 0$.

$x + 1$ means: *optionally x*. This is illustrated by rewriting $(x + 1) \cdot y$ using the given axioms:

$$\begin{aligned} (x + 1) \cdot y &= x \cdot y + 1 \cdot y \\ &= x \cdot y + y \end{aligned}$$

The parallel merge operator $\|$ is defined in terms of the alternative and sequential composition operators. This definition also requires two auxiliary operators:

$$x \parallel y = x \lfloor\lfloor y + y \lfloor\lfloor x + x|y$$

[4] These sections contain some text fragments literally copied or adapted from the predecessor paper. There have been some syntax changes:

- Code fragments now have different flavors of brace pairs.
- All fat arrows related to dataflow, such as => and ==> are now curly arrows such as ~~> and ~~>.
- Script closure notation now uses rectangular brackets (`[....]`) instead of triangular brackets (`<....>`).

[5] This description of ACP has largely been taken from Wikipedia

- $x \lfloor\lfloor y$ - "left-merge": x starts with an action, and then the rest of x is done in parallel with y.
- $x|y$ - "communication merge": x and y start with a communication (as a pair of atomic actions), and then the rest of x is done in parallel with the rest of y.

The definitions of many new operators such as the left merge operator use a special property of closed process expressions with $\cdot$ and $+$: with the axioms as term rewrite rules from left to right (except for the commutativity axiom for $+$), each such expression reduces into one of the following normal forms: $(x+y)$, $a \cdot x$, 1, 0. E.g. the axioms for the left merge operator are:

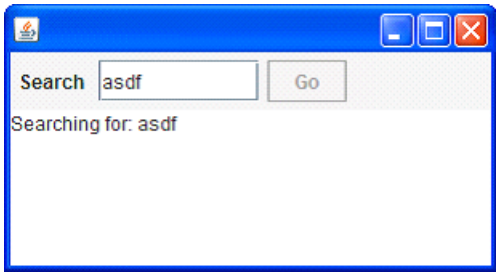
$$\begin{aligned}(x+y)\lfloor\lfloor z &= x\lfloor\lfloor z + y\lfloor\lfloor z\\ a\cdot x\lfloor\lfloor y &= a\cdot(x\lfloor\lfloor y)\\ 1\lfloor\lfloor x &= 0\\ 0\lfloor\lfloor x &= 0\end{aligned}$$

Again these axioms may be applied as term rewrite rules so that each closed expression with the parallel merge operator $\|$ reduces to one of the four normal forms. This way it has been possible to extend ACP with many new operators that are defined precisely in terms of sequence and choice, e.g. interrupt and disrupt operators, process launching, and notions of time and priorities.
Since its inception in 1982, ACP has successfully been applied to the specification and verification of among others, communication protocols, traffic systems and manufacturing plants.
In 1989, Henk Goeman unified Lambda Calculus with process expressions [6]. Shortly thereafter, Robin Milner et al developed Pi-calculus [10], which also combines the two theories.

3. Examples: Simple GUI Applications

Suppose we need a simple program to look up items in a database, based on a search string.

The user can enter a search string in the text field and then press the Go button. This will at first put a "SearchingÉ" message in the text area at the lower part. Then the actual search will be performed at a database, which may take a few seconds (simulated by a call to Thread.sleep). Finally the results from the database are shown in the text area.



If you would program this functionality in plain Scala, the resulting code would be like:

```
val searchButton = new Button("Go")     {
  reactions += { case ButtonClicked(b) =>
    enabled = false
    outputTA.text = "Starting..."
    new Thread(new Runnable {
     def run() {
      Thread.sleep(3000)
      SwingUtilities.invokeLater(new Runnable{
        def run() {outputTA.text="Ready"
                   enabled = true
      }})
    }}).start
  }
}
```

Here `outputTA` denotes the output text area. A solution in Java would be similar. This code looks very technical: lots of indentations and braces. The control flow is hidden in nested functions. Parallelism is done by calling a method `start` on a Thread object. This looks like a usual method call, but something magic happens inside. Parallelism does not get a similar basic treatment as statement sequences do.

The order in which the lines are executed is spaghetti-like:

- The first two lines are done during initialization, in the main thread.
- Then a call back block follows, which, executed when the button is pressed. Disabling the button and setting the "Starting..." text must be done in the Swing thread; this happens to be the case with the call back, so no special provision needs to be taken.
- The call to `start` makes a background thread start that will execute a sleep
- After this sleep, the background thread schedules code for execution in the Swing thread, to set a "Ready" text and to enable the button.

Between the static program text and the dynamic process is a rather large conceptual gap. The programming task is hard and boring. The result: many applications fail to appropriately enable and disable their GUI widgets, or they are not responsive, or they even hang every now and then. This situation is unnecessary.
The SubScript notation is much more concise and intuitive:

```
live = searchButton
       @gui: {outputTA.text="Starting..."}
       {* Thread.sleep(3000) *}
       @gui: {outputTA.text="Ready"}
       ...
```

The new lines here denote sequential composition. There is also a semicolon to denote sequences. This is much like the situation in Scala.[6]

- Line 1: `live` is a method like refinement called "script" for the controller behavior. `searchButton` is an object that is silently converted into a script call `clicked(searchButton)`, using Scala's support for implicit conversions. This call "happens" when the user presses the search button. As a bonus, the script makes sure the button is exactly enabled when applicable, i.e. when the program is ready to handle a button click.
- Lines 2 and 4 each write a message in the text area. An annotation, `@gui:`, makes sure this happens in the Swing thread, as needed.
- Line 3 simulates the lasting database search using a sleep call. The asterisks next to the braces specify that this is done in a background thread, so that neither the GUI nor the main thread will be blocked meanwhile.
- Line 5 turns the foregoing into an "eternal" sequential loop (`...`, "etcetera") of search sequences

SubScript cooperates well with Swing: the programmer can easily specify event handling, widget enabling, and switching to the GUI thread. This is not due to specific language features, but through a custom Swing compatibility layer, which indicates that SubScript may also conveniently be adapted to other frameworks.

You may write the code using more refinements; e.g.

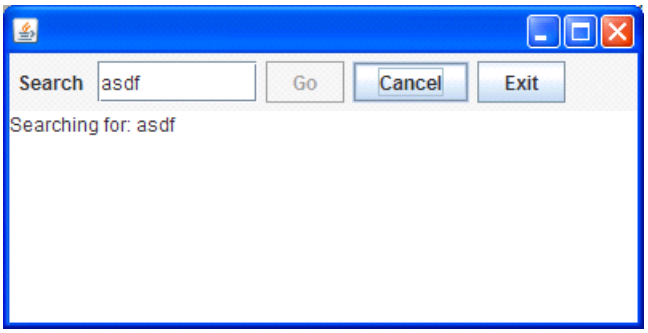
```
live              = searchSequence...

searchSequence    = searchCommand
                    showSearchingText
                    searchInDatabase
                    showSearchResults

searchCommand     = searchButton
```

etc.

3.1 Extending the program

Now we add some realistic requirements to the program.

- The search action may also be triggered by the user pressing the Enter key in the search text field.
- The search action requires that the input text field is not empty; only then should the search button be enabled
- The user should be able to cancel an ongoing search, by clicking a Cancel button, or pressing the Escape key.
- As long as the database search is ongoing, the progress should be indicated: 4 times per second a number is appended to the output text area
- The user can exit the application by clicking an Exit button, or by clicking in the close box at the window's upper right corner. But exiting should first be confirmed in a dialog box.



In a Java or Scala version the application state would need to be kept in variables; updating these would be nontrivial. The progress indicator would be cumbersome and error-prone to program (and that is why it is rarely present). It is easier to grow the SubScript version.
The 3 user commands will be:

```
searchCommand = searchButton + Key.Enter
cancelCommand = cancelButton + Key.Escape
exitCommand   = exitButton   + windowClosing
```

The first and second plus operators create exclusive choices between buttons and key codes. These operands are not processes, but data items for which implicit conversions to processes have been defined (such as `clicked` and `keyPressed`)[7].
Script `windowClosing` acts on window closing events; it is defined in `SubScript.swing.Scripts`.
Exiting is implemented using a process named `exit` that runs in or-parallel composition to the rest. The or-parallel operator is `||`, it means that all operands happen; as soon as one finishes successfully then the other is terminated and the whole composition terminates successfully. In this case, the left hand operand is an eternal loop of search sequences; the right hand operand is a (probably) finite loop.

The `exit` process starts with the exit command being given; then a confirmation dialog is run; all to be repeated while the result of the confirmation dialog is false. The result of the confirmation dialog is transferred using a dataflow operator

[6] SubScript has a similar semicolon inference as Scala. There are some difference. Like many behavior operators, sequential composition in SubScript has 2 or more operands, which is relevant for loops. Also, inferred semicolons have a slighter lower priority than explicitly specified semicolons.

[7] We can combine this way any kind of item; the implicit conversions allow for an algebra of general items rather than just of processes.

to a while construct; this operator is a curly arrow that names and types the flowing data item.

```
live = searchSequence... || exit

exit = exitCommand
       @gui: {! confirmExit !}
                ~~(b:Boolean)~~> while(!b)
```

The @gui annotation in combination with special brace pairs around `confirmExit` ensure that the dialog is run asynchronously in the GUI thread; this way other parts of this program may remain active.

For the search sequence we now add items at the start and the end.

`searchGuard` is an "active guard" containing a sequential loop. It first checks whether the text field contains some text. If so, there is an "optional break", specified by the dot. This means that the sequence and thus also the guard may end successfully, so that `searchCommand` becomes active. However maybe an event happens at the text field before the user issues this search command; then the check needs to be redone, etc (`...`).
Between `searchGuard` and `searchCommand` is a space. Like in Scala, this construct has a high priority, but unlike in Scala, it denotes sequential composition, in addition to semicolons and new lines.
After `searchCommand` a new line follows; this separates the first line from the remaining five lines. Therefore the rest, including cancelSearch, can only become active after the `searchCommand` has happened. `cancelSearch` is preceded by a slash symbol, which stands for disruption: the left hand side happens, possibly disrupted when the right hand side starts happening. The parentheses group the items on the preceding lines, so that the whole becomes the left hand side of the slash operator.

```
searchSequence = searchGuard searchCommand
                 ( showSearchingText
                   searchInDatabase
                   showSearchResults
                 )
                 / cancelSearch

searchGuard     = if(!searchTF.text.isEmpty) .
                   anyEvent(searchTF)
                   ...

cancelSearch = cancelCommand showCanceledText

showSearchingText = @gui: {outputTA.text =...}
showSearchResults = @gui: {outputTA.text =...}
showCanceledText  = @gui: {outputTA.text =...}
```

The database search was mimicked by a few seconds of sleeping; this gets a `progressMonitor` process in an or-parallel combination.
This `progressMonitor` is an eternal loop of:
wait 1/4 second;
then append a loop counter to the output text field.
The pseudo-value `here` denotes "the current operand"; it is comparable to `this`, the "current object". Its field `pass` yields 0, 1, 2, ... in subsequent passes of the loop.

```
 searchInDatabase  = {*Thread.sleep(3000)*}
                   || progressMonitor

progressMonitor   = {*Thread.sleep(250)*}
                    @gui:{searchTF.text
                                  += here.pass}
                    ...
```

4. SubScript Features

SubScript extends Scala with a construct named "script". This is a counterpart of ACP process refinements, that coexists with variables and methods in classes. The body of a script is an expression like the ACP process expressions.

4.1 Notation

Esthetically, ACP processes are preferably notated with the mathematical expression syntax. However, ACP symbols ·, ‖ are hard to type; for a programming language ASCII based alternatives are preferred. SubScript therefore applies a semicolon (`;`) and ampersand (`&`) for sequential and parallel composition. The special ACP processes 0 and 1 would clash with the usual notation for numbers; therefore these are replaced by symbols: `(-)` and `(+)`.
As with multiplication in math, the symbol for sequence may also be omitted, but then some white space should separate the operands. As usual, the semicolon should have low precedence, and the white space operator should have high precedence. This way one can get rid of parentheses. Instead of `(a;b)+c; d` and `(a b + c) d` one could write `a b + c; d`.
Scripts are usually defined together in a section, e.g.,

```
 script..
   hello =          {! print("Hello,") !}
   test  = hello &  {! print("world!") !}
```

From here on the section header `script..` is mostly omitted for brevity.

4.2 Scala Code Fragments

`{! print("Hello,") !}` is a fragment of Scala code that corresponds with an atomic action in the sense of ACP. There are other flavors of brace pairs that have different execution modes and different correspondence with ACP

actions. Often the start and end of such fragments correspond with atomic actions in ACP. This way code fragments may overlap, which is useful when they are run in distinct threads, or when they denote actions that take some simulation time in a discrete event simulation context.

`{!...!}`	normal code fragment: by default a single atomic action, executed in the main thread
`{*...*}`	code executed in a new thread; start and end correspond with atomic actions
`{.....}`	a single atomic action, executed by an event handler
`{.....}`	sequence of atomic actions, executed by an event handler
`{:...:}`	a "tiny" code fragment, executed in the main thread, not corresponding with an atomic action

Within code fragments , Scala expressions may use a special value named `here`. It refers to the current node in the call graph, like `this` refers to the current object. `here` is in particular useful for implementing event handling scripts. For convenience it is an implicit value so that it may be left out of parameter lists containing an implicit formal parameter having the node type.
In case an uncaught exception happens inside the Scala code of a (pair of) atomic actions, the action will fail, rather then succeed. It is as if it ends in the 0 process, or, in ACP terms, as $a \cdot 0$. Note the equivalence $a = a \cdot 1$, which suggests that normally an atomic action ends in success.

4.3 Annotations

An annotation is a piece of Scala code that is executed when the annotated part of a program is activated. The code may refer to its operand using the value named `there`, which is implicit, instead of `here`. It may in turn register callback code for other events that happen on the operand, e.g. when it is deactivated. This was applied for automatic GUI widget enabling and disabling, as seen in the previous examples.
Annotations can also change the execution behavior for code fragments. E.g. in

```
clearText = @gui: {! aTextField.text = "" !}
```

the code fragment will be executed asynchronously in the Swing GUI thread.
This may take some time; meanwhile other code fragments may be executed. Therefore the code fragment will now correspond with 2 atomic actions, like with the code fragments that are executed in new threads.
When combined with a tiny code fragment the annotation will execute the code synchronously in the Swing GUI thread.

4.4 Parallelism

Between `hello` and `{!print("world!")!}` is a parallel operator: `&`. Each operand essentially contains a simple code fragment rather than code to be run in a separate thread. Therefore one operand will be executed before the other; the result is either "Hello, world!" or "world!Hello,". In general the atomic actions in concurrent processes are shuffle merged, like one can shuffle card decks.
In the "hello world" example the most straightforward execution strategy will deterministically apply a left-to-right precedence for the code fragments that are operands to the parallel operator `&`. However, alternative strategies are possible, e.g. for random simulations.
SubScript offers more forms of parallelism. Most notably the operator `||` denotes *strong or-parallelism*: it succeeds when any operand has success; it terminates when any operand terminates successfully.

4.5 Control and Iteration

SubScript has `if-else` and `match` constructs like the ones in Scala. Six operand types support iterations and breaking:

`while`	marks a loop and a conditional mandatory break
`for`	much like `while` and like the Scala for-comprehension
`...`	marks a loop; no break point, at least not here
`..`	marks a loop, and at the same time an optional break
`.`	an optional break point
`break`	a mandatory break point

4.6 Method Calls and Script Calls

The body of the script `test` contains call to script `hello`. These are much like method calls.
A SubScript implementation will translate each script into a method. This way most Scala language features for methods also apply to scripts: scripts may have both type parameters and data parameters; each parameter may be named or implicit. Variable length parameters and even script currying are possible.

If a tiny code fragment contains just a method call then the special brace pair may be omitted. Thus the following two fragments are equivalent:

```
{:print("world!"):}
 print("world!")
```

4.7 Result Values

Code fragments and script have result values, which are comparable to method return values. A difference is that a

method returns only once, whereas the script result value is available to the caller each time that the script has a success; this may be more than once, due to the 1-element of ACP. The following scripts each have result type `Int`:

```
s1:Int = {:5:}
s2 = s1
s3 = s2; {!5!}^
```

The first script has its result type explicitly stated; for the others the type is inferred. The following rules apply for script results: `Script[T]`

- If the script expression contains code fragments or script calls that are suffixed by a caret symbol (`^`) then the result value of the script is set to the result of such a code fragment or script call, at moments when those have success. If the script result type has not been specified explicitly then the result type is the union type of the types of those code fragments and of the result types of those script calls.
- If there is exactly one code fragment or script call in the script then it is considered to have such a caret suffix even if it is not explicitly present.
- In other cases the result type of the script is `Nothing`.

In `{5}^v` the result of the code fragment is stored in a local variable named `v`. As no declaration for this name was in scope, the fragment also implicitly declares the variable here.
Result values are packed in a `Try` container, which is either a `Success` or `Failure` container. The latter may hold an exception, which may come of use when an exception has happens inside a code fragment.

4.8 Script Lambdas

Script expressions placed between rectangular brackets, such as `[{:5}]` and `[(a;b)+c; d]`, are values of type `Script[T]` for an inferred type T. These are essentially parameterless script lambda's (AKA closures); a variable or value holding such a lambda may act inside a script expression as a script call.
The Scala way of defining parameterized lambda expressions applies as well, essentially giving parameterized script lambda's, e.g.,
`(i:Int) => [(a;b)+c; d]`

4.9 Script Execution from Scala

From Scala code a so called *script executor* may execute a script lambda, as in

```
executor.run([test])
```

The executor may be tailored for the type of application, e.g. discrete event simulations. After the execution ends, the executor may provide information on the execution, e.g. on whether the script ended successfully or as deadlock (δ).

The DSL method `_execute` creates a fresh CommonExecutor (the default executor type) and then calls its run method with the script closure, e.g.,

```
subscript.DSL._execute([test])
```

Other types of executors could be more suited for specific application domains, such as discrete event simulations and multicore parallelism.

5. Call Graph Semantics

A SubScript program is executed by an executor that maintains a call graph, which is at run time derived from the static structure of the invoked scripts.

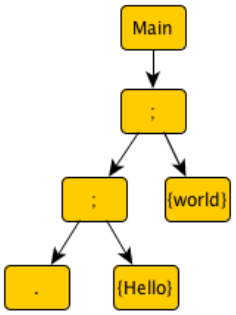


Figure 1. Template Tree

This static structure may be represented by so-called "Template trees". For instance, consider the following process which prints optionally "Hello", and then "world!":

```
Main = . {! print("Hello ") !};
         {! print("world!") !}
```

Figure 1 gives the template tree. At run time this tree is used to grow and prune the call graph, as shown in figure 2. The predecessor paper contains a more detailed explanation.

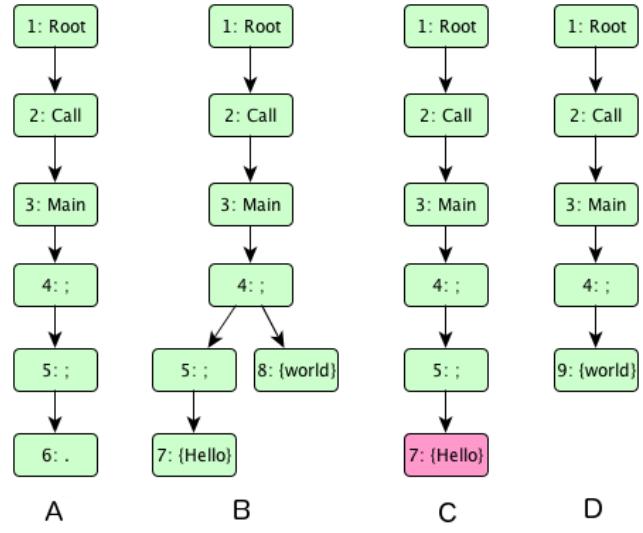


Figure 2. Call Graphs

6. Basic Implementation

At first SubScript had been implemented as a domain specific language (DSL); the so called SubScript Virtual Machine executes scripts by internally doing graph manipulation. The VM has been programmed using 2500 lines of Scala code. This is not a complete implementation; most notably support for ACP style communication is still to be done. When complete the VM may contain about 4000 lines. In principle the DSL suffices for writing the essence of SubScript programs. However, with the special syntax, e.g. for parameter lists, n-ary infix operators, various flavors of code fragments, specifications become considerably smaller and these require much less parentheses and braces (which is also important for clarity).

A special branch of the Scala compiler was modified so that it translates the genuine SubScript syntax to the DSL. This took about 2000 lines of Scala code, mainly in the scanner, the parser and the typer.

7. Dataflow Programming

A relatively new SubScript language feature is dataflow, expressed by curly arrows as seen in the Gui controller example:

```
exit = exitCommand
       @gui: {! confirmExit !}
                  ~~(b:Boolean)~~> while(!b)
```

We could also have specified a smaller version of the dataflow operator, that does not give a name and type to the flowing data item:

```
exit = exitCommand
       @gui: {! confirmExit !} ~~> while(!_)
```

Now the `while` in the right hand side has an underscore as parameter; like in Scala it denotes a default parameter, and it turns its close environment into a parameterized lambda. The left hand operand is also a lambda, so that it has its own result value.

The dataflow construct is a kind of sequential composition, a difference being that it cannot become a loop.

A similar construct lets exceptions flow. E.g. in `x ~/~> y` when `x` ends in failure (without success), `x`'s result is a `Failure` wrapper containing either an exception or null. Then `y` is executed with the flown exception or null as a parameter.

Such dataflow and exception flow may be combined in a ternary operator:

```
x ~~> y +~/~> z
```

This starts with `x`. When `x` has success, `y` is activated with `x`'s normal result value. When `x` terminates as a failure, `z` is executed with `x`'s resulting exception.

Similar variations are possible for dataflow operators with named items, so that these become analogous to a combination of match statements and exception handers, e.g.,

```
x ~~(b:Boolean     )~~> y1
 +~~(i:Int if i<10 )~~> y2
 +~~(  _           )~~> y3
 +~/~(e:IOException)~~> z1
 +~/~(e:  Exception)~~> z2
 +~/~(e:  Throwable)~~> z1
```

7.1 Example: Twitter Search

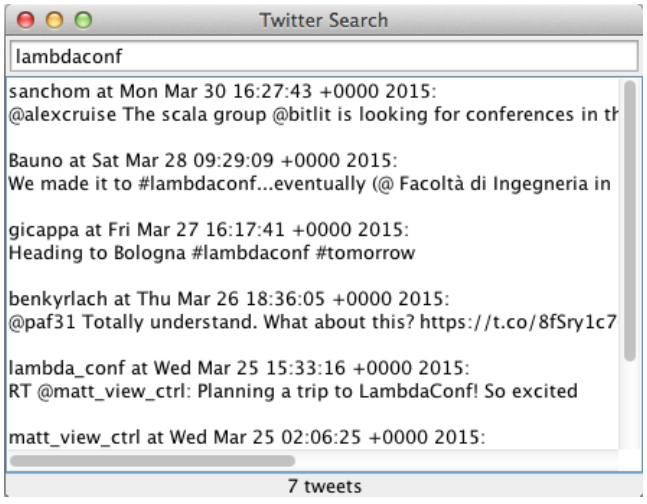


Figure 3. Twitter Search application

A simple Twitter search application contains an input text field and a result text area; when the user has changed the content of the input text field the application starts a request to the Twitter web service to get 10 tweets matching the input text. But Twitter imposes request rate limit on its API, and the client should not exceed this. Therefore after each change in the text field the application waits 200 milliseconds before sending the request to Twitter. If meanwhile the text field changes again, we will restart the wait. When the input text changes while a request had already been sent and the result was awaited, then that process is disrupted as well.

The searches may go wrong; we can (intentionally) send an empty search string, which will result in an error reply by the Twitter server.

A pure Scala version for the controller would contain something like:

```
def bindInputCallback = {
  listenTo(view.searchField.keys)

  val fWait   = InterruptableFuture {...}
  val fSearch = InterruptableFuture {...}

  reactions += {case _ =>
   fWait.execute()
        .flatMap {case _ => fSearch.execute()}
        .onComplete{
          case Success(tweets) =>
                      Swing.onEDT{...}
          case Failure(e:Throwable) =>
                      Swing.onEDT{...}
} } }
```

InterruptableFutures are a flavor of futures that can be cancelled on demand. This functionality requires a bunch of ad-hoc utility code in pure Scala, whereas it is supported out-of-the-box in SubScript, backed by theory.
The SubScript version has a live script for the controller, containing a loop of complete search sequences.

```
live    = initialize; (mainSeq/..)...

mainSeq = anyEvent(view.searchField)
          {* Thread sleep keyTypeDelay *}
          {* searchTweets *}
            ~~(ts:Seq[Tweet])~~>updateView(ts)
          +~/~(t: Throwable )~~>setErrorMsg(t)

updateTweetsView(ts: Seq[Tweet]) = @gui: {...}
setErrorMsg     (t : Throwable ) = @gui: {...}
```

The slash and the two dots in `mainseq/..` denote a disruptive loop that starts by activating 1 instance of `mainSeq`. As soon as the first atomic action therein happens (anyEvent in the search field) a next iteration of the disruptive loop is activated. Thus if a next anyEvent arrives soon enough, before the rest of the ongoing earlier `mainSeq` instance has terminated successfully, that ongoing instance is disrupted and a new delay starts, and a new instance of `mainSeq` is activated, etc. The disruptive loop ends when such a `mainseq` has terminated successfully.

It is also possible to use futures in the script. Suppose we have

```
def delay  = Future{Thread sleep keyTypeDelay}
def search = Future{searchTweets}
```

And suppose an implicit conversion from futures to scripts is in scope. Then we can use the futures as follows:

```
mainSeq = anyEvent(view.searchField)
          delay
          search
            ~~(ts:Seq[Tweet])~~>updateView(ts)
          +~/~(t: Throwable )~~>setErrorMsg(t)
```

7.2 Implementation

Arrows with parameters are syntactic sugar, e.g.,

```
x ~~(b:Boolean     )~~> y1
 +~~(i:Int if i<10 )~~> y2
 +~~(  _           )~~> y3
 +~/~(e:IOException)~~> z1
 +~/~(e:  Exception)~~> z2
 +~/~(e:  Throwable)~~> z1
```

desugars into

```
x ~~> (case b: Boolean     ==> y1
       case i: Int if i<10 ==> y2
       case  _             ==> y3)
+~/~> (case e: IOException ==> z1
       case e:   Exception ==> z2
       case e:   Throwable ==> z1)
```

Here `==>y` is sugar for `=>[y]`.
In an expression of the form `x ~~> y +~/~> z`, all three operands are lambdas, so they have their own result types. Let X, Y and Z be the result types of x, y and z, and let T be the most specific ancestor type of Y and Z. Then the expression is transformed into [8]

```
var xResult: Try[X] = null
do @{there.onSuccess{xResult=there.$}}: x
then y(xResult.get)^
else z(xResult match {case Failure(f) => f
                      case null    => null})^
```

`$` is a `Try`: it either holds the result value of a node packed in a `Success` wrapper, or a `Failure` holding an exception or null.
`do x then y else z` is a ternary construct meaning: do `x`; after `x` has success `y` is activated; in case `x` ends in failure (i.e. deactivates without success) then `z` is activated.
Both the then-part and the else part are optional, but not at the same time. If the else-part is absent, `do-then` is a binary sequential operator; it cannot iterate, so if it has an iterator operand then that affects an operator such as `;` or `&` higher in the call graph.
Expressions of the form `x ~~> y` and `x ~/~> y` are treated similarly.

[8] This is a minor simplification. In the presented code the declaration of variable `xResult` implies a sequential composition, which is in general undesirable. This has been overcome using slightly more technical code. Probably an alternative for this sequential composition would be appropriate, e.g., a `let ... in ...` construct as known from functional programming languages such as ML

8. SubScript Actors

SubScript has a message handling feature to support the specification of Akka actor behavior. E.g. the following pair of actors would exchange some "Ping" messages, terminated by "Stop":

```
class Ping(another: ActorRef) extends Actor {
  another ! "Ping"
  another ! "Ping"
  another ! "Stop"
}

class Pong extends SubScriptActor {
 script
    live = <<"Ping">> ... / <<"Stop">>
}
```

The brackets <<....>> denote a message handler; on the inside there is essentially a partial function, like in the usual `receive` methods of actors. Multiple of such message handlers may be active at the same time; these handlers are considered when Akka gives an message to the actor for handling. If any such active message handler has a partial function that is defined for the message, it will handle it, and the actor's `receive` method will not be called.

Message handlers may be more complicated than the presented ones. The general form is:

```
<< case p_1 => scalaCode_1 ==> scriptExpr_1
   case p_2 => scalaCode_2 ==> scriptExpr_2
   ....
   case p_n => scalaCode_n ==> scriptExpr_n
>>
```

This is much like a Scala partial function that is usually returned by an actor's `receive` method. The long arrow with `scriptExpr` is new. This specifies the behavior that holds after the corresponding message handling. Because << and >> act as a bracket pair, the names of parameters, values and variables before the long arrow are available in the `scriptExpr`. It may also safely use `sender`, which is a local value silently copied from the class variable with the same name.

Simpler forms are possible because of the next rules:

- The sections that are proceeded by the arrows may be omitted.
- If there is only one case then that tag may be omitted.

8.1 Data Store example

For a more advanced example, imagine a data store with a proxy and a backend. A client may send information requests to the proxy. This sends the request on to the backend store. The latter will reply the requested information. Based on this, the proxy sends a new request for detailed information to the backend store. After the proxy has received this detailed information it sends all received information back to the original client.

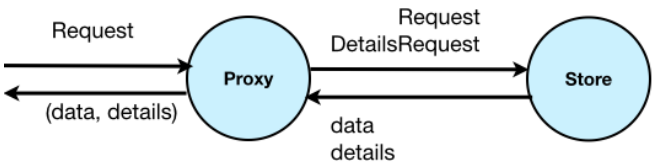


Figure 4. Data store with proxy

In plain Scala+Akka a state machine would typically specify the behavior of the data store proxy:

```
class DataProxy(ds: ActorRef) extends Actor {

  def waitingForRequest = {
    case req: Request =>
      ds ! req
      context become waitingForData(sender)
  }

  def waitingForData(requester: ActorRef) = {
    case data: Data =>
      ds ! DtlRequest(data)
      context become
          waitingForDetails(requester, data)
  }

  def waitingForDetails(requester: ActorRef,
                        data: Data) = {
    case dtls: Details =>
      requester ! (data, dtls)
      context become waitingForRequest
  }
}
```

The states have meaningful names, but the control flow may be confusing since the program seems to jump between these states.

A SubScript solution may apply nested message handlers:

```
live = << req: Request
       => val client = sender; ds ! req
      ==>
         << data: Data
         => ds ! DtlRequest(data)
        ==>
            << dtls: Details
            => client ! (data,dtls)
            >>
         >>
       >>
       ...
```

This is shorter than the plain Scala solution. The ability to nest message handlers is powerful, but that comes with a price: like with regular control structures, each deeper level makes the code more complex, in general.

With dataflow operators the code may be flattened and shortened:

```
live =
  << req : Request ==> {ds ? req}
  ~~(data: Data   )~~> {ds ? DtlRequest(data)}
  ~~(dtls: Details)~~> {:sender!(data,dtls):}
  >>
  ...
```

Here the message to the backend store are sent using the `?` operator: in Akka this sends a message using a new, short-lived actor, and asynchronously awaits the answer from the addressee. The `?` operator returns a future that completes when the answer has been received. These future value expressions are enclosed inside normal brace pairs, which do not denote actions but just Scala values. Just like with `delay` and `search` in the Twitter example, these futures are converted using an implicit script. The brace pairs are needed to parse these as Scala expressions rather than process expressions.
`{:sender!(data,dtls):}` is a tiny code fragment that sends the data including the details back to the original sender of the request. In a plain Akka actor `sender` is a class variable that gets overridden each a new message arrives. Here inside the message handling brackets, «...», that class variable is copied into a local script variable with the same name, so the name `sender` may be safely be used later inside the message handler.

8.2 Implementation

The Akka framework provides convenient aspect oriented programming hooks that facilitate a compatibility layer for SubScript.
In particular trait `akka.actor.Actor` has methods `aroundPreStart`, `aroundPostStop` and `aroundReceive`.
SubScript Actors are implemented in a separate trait, named `SubScriptActor` which extends `akka.actor.Actor` and which implements these hook methods.
Normally the SVM executes a script called from Scala in a loop that handles call graph message synchronously in sequence. The loop's body is a call to a method named `tryHandleMessage`; this method may as well be called from outside. This gives 3 options for the coexistence of Akka and the SVM for running SubScript actors:

- There are 2 private thread to run these; these are synchronized using calls to `wait` and `notify`.
- A separate SubScript process that runs in its own SVM coordinates Akka and the other SVM
- Timer callbacks and the `aroundReceive` hook call the method `tryHandleMessage` in the SVM.

Currently the third option is taken.
`aroundPreStart` is called by Akka when an actor starts; it starts the actor's lifecycle behavior within the SubScript virtual machine (SVM):

```
override def aroundPreStart() {
  runner.launch([lifecycle])
  super.aroundPreStart()
}
```

The `runner` is of type `SubScriptActorRunner`, a trait with methods to launch and execute scripts. Its working implementation is an object `SSARunnerV1` - this executes the SubScript scripts under supervision of Akka's scheduler, rather than from a simple `while`-loop that normally runs in a ScriptExecutor.
The actor's `lifecycle` script is:

```
lifecycle = live || terminate; die
```

This starts two processes in parallel:

- Process `live` is abstract in this trait; in a subclass it should specify the behavior of the actor, such as what messages can it handle and when.
- Process `terminate` is a means to terminate the actor on demand. Internally it has an event handling code fragment; this will be executed when the actor is stopped, using the hook `aroundPostStop` that Akka provides.

After either `live` or `terminate` has completed, the `die` process starts. Its job is to do the final clean-up: it unloads the actor from the `ActorSystem` and the memory.
Whereas a `akka.actor.Actor` only uses the `receive` method for message handling, a `SubScriptActor` may have multiple message handlers active:

```
private val callHandlers = ListBuffer[
    PartialFunction[Any, Unit]]()
```

The SubScript compiler translates a message handler

```
<< case p_1 => scalaCode_1 ==> scriptExpr_1
   case p_2 => scalaCode_2 ==> scriptExpr_2
   ....
   case p_n => scalaCode_n ==> scriptExpr_n
   >>
```

into the following script call:

```
r$(case p_1 => scalaCode_1; [scriptExpr_1]
   case p_2 => scalaCode_2; [scriptExpr_2]
   ...
   case p_n => scalaCode_n; [scriptExpr_n])
```

So the parameter of this call is a partial function which returns a script lambda. In case a long arrow with a `subScriptExpr` is absent, `null` is returned.
The main task of the script `r$` is to handle an event that represents a message being accepted; and then make the related `subScriptExpr` happen. This requires some technical preparation:
An annotation derives a new partial function from the given one, by appending code that stores the returned script lambda and causes the event to happen. This appended handler is registered and later unregistered, in the same way as a script for GUI controllers enables and disables a button.

```
def script r$(handler:
   PartialFunction[Any, ScriptNode[Any]])
= var s:ScriptNode[Any]=null
  @{val h = handler andThen
    {hr => {s = hr; there.eventHappened}}
    synchronized {callHandlers += h}
    there.onDeactivate {synchronized
                {callHandlers -= h}}
  }:
  {.  .}
  if s != null then s
```

When a message arrives for the actor, Akka will likely call its `aroundReceive` method. This first yields control temporarily to the `SubScriptActorRunner` which in turn calls `tryHandleMessage` repeatedly to let the SVM maintain the script call graph. Thereafter `aroundReceive` tries the `callHandlers` for one that is capable to handle the received message (by calling `isDefinedAt`). If no such handler exists, the message processing falls back to the regular `receive` method.

9. Interoperation with Futures

Scala's `Futures` and SubScript processes are fundamentally similar. They both represent a potentially time-consuming operation that results in some value (success) or in an exception (failure). However, while a future completes once, a script may have success more than once, and a failure may even occur after a success, like in the ACP process $1 + a \cdot 0$. Also, a future is in general not cancelable whereas a script is inherently cancelable, e.g. in contexts of the operators `&&`, `||` and `/`.
It is possible to specify a future as a kind of script call in a script expression, using an implicit conversion script:

```
implicit script future2script[T]
           (f:Future[T]): T
= @{var isExcluded = false
    there.onExclude{isExcluded=true}
    f.execute()
     .onComplete{
       case aTry if !isExcluded =>
            there.executeForTry(aTry)
     }
  }:
  {. .} // empty event handling code fragment
```

`there.onExclude` registers a callback at a node which executes when the node is being excluded, e.g. when an atomic action happens in a different branch of the + operator. In this case the callback makes sure that once the script has been canceled, any later completion of future will have no effect.
Otherwise the completion of the future calls `executeForTry` on the event handling code fragment `{. .}`. This happens to execute an empty code fragment, but the fragment corresponds with an atomic action, and when that succeeds the SVM will grow and prune the call graph further. `executeForTry` also sets the result value of the event handling code fragment to the `Try` value that the future had yielded.
Conversely it is possible to convert a script closure to a future:

```
implicit def script2future[T](s: Script[T]):
                    Future[T] =
{
  val p = new Promise[T]
  val s1 =
    [ @{there.onSuccess{p success there.$}
        there.onFailure{p failure there.
                                    failure}
      s
    ]
  AkkaScriptRunner.execute(s1)
  p.future
}
```

Here `AkkaScriptRunner` is much like the earlier discussed `SSARunnerV1`: it may execute a given script in cooperation with Akka's actor system. During this execution it adds an annotation the script that registers handlers for success and failure: on such occasions that the corresponding values are transferred from the script to the future.
A problem with this conversion is that the script may have multiple times success, so that the promise's success method may be called multiple times. Some extra logic may prevent this, but there is anyway a kind of impedance mismatch. Anyway we don't know yet whether the conversion from scripts to futures will be useful or not.

10. Performance

We measured performance of using a Sieve of Eratosthenes application, in which each prime sieve was implemented by an actor process. On an iMac with a 2.7GHz Intel Core i5, the plain Scala version got a throughput of about 1,000,000 messages per second, whereas the SubScript version only

got about 10,000 messages per second. The performance penalty factor is therefore about 100. We expect to improve performance considerably by relatively small modifications, but we estimate that this way the gap will remain at least a factor 25.

One reason for this can be inefficient mechanics of script processing by the virtual machine. Currently, all the processing relies heavily on message passing within the virtual machine. It is possible, that the a good fraction of such messages are "noise", meaning they don't carry any useful information and just throttle execution speed.

In principle it seems possible that a SubScript virtual machine follows a different execution strategy: the compiler or VM would analyze a behavior specification and transform it into a finite state machine representation. Then the SubScript actors performance could much be much closer to the one of plain Scala actors.

11. Related Work

The predecessor paper contains an overview of other languages that show some resemblance to this work. SubScript seems to be unique in allowing for an process algebra approach to actor programming.

There are some noticeable papers that apply process algebra as a theoretical underpinning to actors: [1], [5] use Pi-calculus, and [12] applies ACP.

The parser combinators library Parboiled [13] offers more flexibility for capturing result values than SubScript currently does. It is a challenge to bring the same expressiveness to SubScript.

12. Conclusion

SubScript offers constructs from the Algebra of Communicating Processes that apply well to reactive programming. It helps to concisely specify internal actor behavior.

Futures may conveniently placed in SubScript process expressions. Likewise SubScript processes may be converted into futures, but there is an "impedance mismatch". A variant of Futures that supports a kind of 1-element from ACP, could be interesting.

Further R&D on ACP, SubScript and Actor systems could focus on design, prototyping, performance analysis, testing and formal reasoning.

SubScript is an open source project[9]. It is currently implemented as a branch of the regular Scala compiler, bundled with a virtual machine and a library for interfacing with Akka actors and Swing GUIs.

There is a performance penalty. Whether that is a problem depends on performance requirements, and on the proportion of CPU time required for internal actions to the total CPU time.

Rather than a branch of the Scala compiler we wish to have a loosely coupled front end. At the time of writing this paper where are therefore developing a Parboiled parser which should produce input for the Scala compiler that invokes Scala macros to perform some specific transformations.

[9] Subscript web site: http://subscript-lang.org

13. Acknowledgement

Anatoliy Kmetyuk received a Google Summer of Code stipend in 2014 for his work on the project.

References

[1] G. Agha and P. Thati. An algebraic theory of actors and its application to a simple object-based language. In *In Ole-Johan DahlÕs Festschrift, volume 2635 of LNCS*, pages 26–57. Springer, 2004.

[2] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335:131–146, May 2005.

[3] P. Ciancarini, A. Fantechi, and R. Gorrieri, editors. *Formal Methods for Open Object-Based Distributed Systems*, volume 139 of *IFIP Conference Proceedings*, 1999. Kluwer. ISBN 0-7923-8429-6.

[4] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, Jan. 1999.

[5] M. Gaspari and G. Zavattaro. An algebra of actors. In Ciancarini et al. [3]. ISBN 0-7923-8429-6.

[6] H. Goeman. Towards a theory of (self) applicative communicating processes: A short note. *Inf. Process. Lett.*, 34(3): 139–142, 1990.

[7] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[8] C. Hoare. Communicating sequential processes. *ACM Computing Surveys*, 7(1):80–112, 1985.

[9] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part i. *I AND II. INFORMATION AND COMPUTATION*, 100, 1989.

[11] A. van Delft. Dataflow constructs for a language extension based on the algebra of communicating processes. In *Proceedings of the 4th Workshop on Scala*, SCALA '13. ACM, 2013.

[12] Y. Wang. Fully abstract game semantics for actors. *CoRR*, abs/1403.6563, 2014.

[13] P. Wills. No more regular expressions. Scala Exchange, London, 2014. Skills Matter.