Declarative Programming with Algebra

Andre van Delft and Anatoliy Kmetyuk

Independent Researchers
in Rijswijk, the Netherlands, and Odessa, Ukraine
andre dot vandelft at gmail dot com
anatoliykmetyuk at gmail dot com

Abstract. The Algebra of Communicating Processes (ACP) is a theory that views sequences and choices as mathematical operations: multiplication and addition. Based on these base constructs others are defined, such as parallel merge, interruption and disruption.
Conventional programming languages may be enriched with ACP features, to gain declarative expressiveness. We have done this in SubScript, an extension to the Scala language. SubScript has high level support for sequences, choices and iterations in a style similar to parser generator languages. It also offers parallel composition operations, such as and- and or- parallelism, and dataflow.
The declarative style is also present in the way various execution modes are supported. Conventional programming languages often require some boilerplate code to run things in the background, in the GUI thread, or as event handlers. SubScript supports the same execution modes, but with minimal boilerplate. It is also easy to compose programs from blocks having different execution modes.
This paper introduces ACP and SubScript; it briefly describes the current implementation, and gives several examples.

1 Introduction

The Algebra of Communicating Processes (ACP) [2] is a concurrency theory that allows for concise specifications of event-driven and concurrent processes. ACP and the related theories CSP [9] and CCS [11] appear to be largely ignored in R&D on declarative programming. This is unfortunate because ACP offers a solid mathematical foundation for reasoning about program behavior, and a uniform approach to high level process compositions such as sequence, choice, parallelism, interruption (a process being suspended while another one executes) and disruption (a process being canceled when another one starts).

It is well possible to program applications using ACP. We are developing an ACP based extension to Scala by the name of SubScript, with process refinements called *scripts*. SubScript contains several constructs and ideas such as or-parallelism, that are not yet covered by ACP; these are listed in [6].

The sequence and choice operators of ACP and Subscript are much like constructs in parser generator languages. SubScript code is therefore much like grammar descriptions. But the style extends to other composition operations such as parallelism, disruption and interruption.

SubScript also supports declarative specification of different code execution modes. In conventional programming languages such as Java, handling events is quite cumbersome: it requires creating, registering and later unregistering event listeners. Other boilerplate code is needed to let things happen in a background thread or in a GUI thread. In SubScript it is possible to largely abstract from this boiler plate. Like in ACP process specifications events to which a process reacts, appear just as actions; similar to internal actions.

It is also straightforward in SubScript to make compositions of code with different execution modes. This is useful for instance in interactive programs. E.g., a recurring pattern for handling user commands is to have a series of the following kinds of actions, that have 3 different execution modes:

- handle an event (e.g. a button being pressed)
- perform an action in the GUI thread (e.g. updating a status label)
- perform an action in the background thread (e.g. requesting data from a web server)
- perform an action in the GUI thread (e.g. showing the results)

SubScript also has anonymous scripts, also known as process lambdas. This comes almost for free from Scala's support for anonymous functions. Using these there is relatively simple syntactic sugar to define a sequential dataflow construct, which happens to be useful for exception handling as well. Another useful feature inspired by Scala is implicit conversion from data to processes.

All of Scala is available in SubScript. This includes concurrency features such as threads, actors and futures; SubScript allows wrapping those on a higher declarative level.

A SubScript implementation is available. It comes with a preprocessor that translates SubScript code into regular Scala code; some specific transformations are deferred to Scala macros. Script translate into methods; their bodies contain calls to the 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: section 2 introduces ACP; section 3 gives two SubScript example applications; section 4 lists language features; section 5 describes a SubScript Virtual Machine; section 6 highlights dataflow programming with SubScript; section 7 dicusses some related work.

The current paper is a follow up to a paper presented at the Scala Workshop 2013 [5] about dataflow programming support in SubScript, with application to actor systems.[1]

2 ACP

The Algebra of Communicating Processes 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[2]. More so than the other seminal process calculi (CCS and CSP), the development of ACP focused on the algebra of processes, and sought to create an abstract, generalized axiomatic system for processes.

[1] This paper contains some text fragments literally copied or adapted from the predecessor paper.

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

ACP uses instantaneous, atomic actions (a,b,c,...) as its main primitives. Two special primitives are the deadlock process 0, also known as δ, and the empty process 1, also known as 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 atomic 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 cannot 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:

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

The primitives 0 and 1 behave much like the 0 and 1 that are usually neutral elements for addition and multiplication in algebra. $x + 1$ means: *optionally* x. This is shown by rewriting $(x + 1) \cdot y$ using the 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:

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

- $x \lfloor\lfloor y$ - "left-merge": first x is to execute an atomic 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:

$$(x + y)\between z = x\between z + y\between z \qquad 1\between x = 0$$
$$(a \cdot x)\between y = a \cdot (x \parallel y) \qquad 0\between x = 0$$

Again these axioms may be applied as term rewrite rules so that each closed expression with the parallel merge operator $\parallel$ 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.

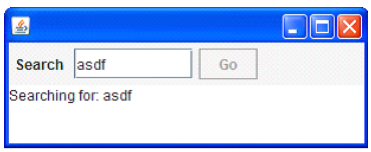
ACP's strict algebraic approach has an advantage over CSP and CCS: this way theorists can study multiple models that satisfy a given set of axioms. This fact was not relevant though choosing ACP as a base for SubScript rather than CSP or CCS. The main reasons were:

– CSP has two choice operators: a deterministic one and a nondeterministic one. This distinction appears unnecessary as CCS and ACP can do without.
– CSP and CCS have *action prefixing*: a kind of sequential composition where the left hand side must be an atomic action (an *event*, in CSP terms); the right hand side cannot be an atomic action. In CCS this is an inconvenient limitation. CSP has a separate sequential composition operator, but also this is an unnecessary complication. ACP treats sequences much like mainstream programming languages do: operands may be atomic, like assignments, or composed, like method calls.

SubScript supports *anonymous processes*, also known as *process lambdas*. These constructs have never been formalized for ACP, but they have been for CCS. In 1989, Henk Goeman unified Lambda Calculus with process expressions[8]. Shortly thereafter, Robin Milner et al developed Pi-calculus [12], which also combines the two theories.

3 Two 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 "Searching" message in the text area at the lower part. Then the actual search will be done 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.

In plain Scala, the required 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. This code looks very technical: lots of indentations and braces. The control flow is hidden in nested functions. Parallelism is done by calling the `start` method 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 are needed.
- 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 not only holds for Scala, but also for almost all imperative languages.

This situation is unnecessary. The SubScript notation is more concise and intuitive:

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

The line breaks here denote sequential composition. [3]

[3] There is also a semicolon to denote sequences. SubScript has a similar semicolon inference for line breaks as Scala.

- 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)`. This is done by an extension of Scala's support for implicit conversions. This call "happens" when the user presses the search button.
- As a bonus, the call to `clicked` 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 programmers can easily specify the GUI controller life cycle, event handling, widget enabling, and switching to the GUI thread. This is not due to specific language features geared towards Swing, but through a custom Swing compatibility layer, with scripts such as `clicked` and methods such as `gui`.

3.1 Extending the program

Now we add some realistic requirements to the program.

- Pressing the Enter key in the search text field triggers the search action as well.
- The search action requires that the input text field is not empty; only then should the search button be enabled
- Clicking button Cancel, or pressing the Escape key cancels an ongoing search.
- 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.
- Clicking an Exit button or in the close box at the window's upper right corner exits the program, provided that the user confirms this in a dialog box.

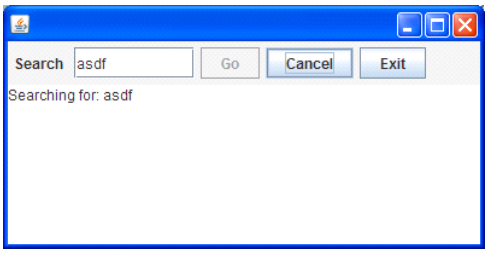
We can start by raising the abstraction level of the code above, giving names to each of its individual actions, so that we can implement these extensions by modifying the definitions of these named actions:

```
live             = searchSequence...

searchSequence   = searchCommand    showSearchingText
                   searchInDatabase showSearchResults

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

In a Java or Scala version the application state would need to be kept in variables; updating these would be non-trivial. 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 three 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`)[4].

The library script `windowClosing` acts on window closing events.

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 both operands execute in parallel; 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 (explained later) 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. The exclamation marks in the brace pairs denote that `confirmExit` is an atomic action in the ACP sense, which is relevant in choice contexts.

The `while` construct at the end does not require the conditional expression to be inside parentheses, as long as it is a simple expression that cannot be confused with the other parts of the script.

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 (`searchTF`) contains some text. If it does, there is an "optional break". This means that

[4] We can combine this way any kind of item for which implicit conversions to scripts are in scope; this yields an algebra of general items rather than just of processes.

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 then break?
                    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; we add a progress monitor process in an or-parallel composition. This `progressMonitor` is an eternal loop: wait a short time and then append a loop counter to the output text field, etc.

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

ACP processes are notated with the mathematical expression syntax. The ACP symbol · for sequential composition is hard to type; therefore SubScript applies a semicolon (`;`) as known from Scala. As with multiplication in math, the semicolon symbol for sequence may also be omitted, but then some white space should separate the operands.[5]

The Scala symbols for and- and or-compositions of booleans, `&`, `&&`, `|` and `||`, were reused for analogous flavors of parallelism in SubScript. Therefore the ACP symbol ‖ corresponds with an ampersand (`&`) in SubScript.

The special ACP processes 0 and 1 would clash with the usual notation for numbers. These are replaced by symbols: `[-]` and `[+]`.[6]

Parentheses in ACP processes are replaced as rectangular brackets in SubScript scripts. This is because parentheses are already heavy in use in the base language Scala: for value expressions, tuple notation and parameter lists.

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 normal fragment of Scala code; by default it is executed in the main thread. Conceptually this corresponds with an atomic action happening in the sense of ACP. This atomic action is relevant for instance in a choice context such as `{! print("Hello,") !} + {! print("world!") !}`

Here as soon as the atomic action happens in the left hand side operand of the plus, the right hand side is excluded: its code fragment cannot be executed any more, and it is marked for deactivation.

There are different flavors of code fragments (`s` means some Scala code):

- `{! s !}` - normal code fragment; corresponds by default with one atomic action.
- `{* s *}` - code executed in a new thread; corresponds with two atomic actions.
- `{. s .}` - a code fragment executed by an event handler, e.g. a GUI listener; corresponds with an atomic action.
- `{... s ...}` - a code fragment that may be executed multiple times by a permanent event listener; each execution corresponds with an atomic action.
- `{: s :}` - a "tiny" code fragment. It does not correspond with an atomic action; therefore it is efficiently executed. Apart from the code being executed, this behaves neutrally in the ACP sense: it corresponds with 0 or 1; which one of these depends on nearest ancestor n-ary operator.

[5] In general Scala's operator precedence rules are followed, except for the dataflow operators; in Scala white space denotes function application; in SubScript it is sequential composition.

[6] Library scripts that refine into such special processes, may be more readable. For the time being we want a minimal set of new keywords.

Normal code fragments may be manipulated to run in a distinct thread such as the GUI thread. In such cases there is a correspondence to two atomic actions instead of one: one atomic action happens just before the start of the code fragment execution, and one happens just after the end. The latter action will not happen when the executing code fragment had been disrupted, e.g. from the disruption operator `/`.

Threaded code fragments run in new threads; they also correspond with two such atomic actions. When disrupted while running, the thread will get an interrupt signal.

Scala expressions within code fragments may use a special value named `here`. It refers to the current node in the call graph (i.e. a generalization of a call stack, see section 5), like `this` refers to the current object. `here` is in particular useful for implementing event handling scripts.[7]

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`. The code 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 tiny code fragment will be executed synchronously in the Swing GUI thread, using the Swing method `SwingUtilities.invokeAndWait()`.

When combined with a normal code fragment the annotation will execute the code asynchronously in the Swing GUI thread using `SwingUtilities.invokeLater()`; meanwhile other code fragments may be executed.[8]

4.4 Parallelism

For each of the boolean operators `&`, `&&`, `|` and `||` there is a counterpart parallel operator in SubScript: `&` and `&&` are and-like; they succeed when all operands succeed. `|` and `||` are or-like; they succeed when any operand succeeds.

`&` and `|` terminate when all operands terminate. `&&` denotes *strong and-parallelism*: it terminates when any operand terminates without success. `||` denotes *strong or-parallelism*: it terminates when any operand terminates successfully.

Between `{!print("hello!")!} & {!print("world!")!}`, 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 parallel branches are shuffle merged, like one can shuffle card decks.

The most straightforward execution strategy will deterministically apply a left-to-right precedence for the code fragments that are operands to the operator `&`. However, alternative strategies are possible, e.g. for random simulations.

[7] For convenience `here` is an implicit value so that it may be left out of parameter lists that have an implicit formal parameter of the node's type.

[8] In annotations `there` is implicit instead of `here`. Thus `@gui:` is equivalent to `@gui(there)`.

4.5 Disruption and Interruption

The slash operator denotes disruption: in `x/y`, both operands are activated; `x` is terminated as soon as an atomic action in `y` happens. For interruption there are two operators: in `x%/y` execution of `x` is suspended as soon as an atomic action in `y` happens; it may resume when `y` has success. The operator `%/%/` is for zero or more interruptions.[9]

4.6 Control and Iteration

SubScript has `if-then-else`, `match`, `while`, `for` and `break` constructs much like counterparts in Scala. The latter three are not limited to sequential contexts, so they enable alternative and parallel iteration control. Some special processes are:

- `break?` denotes an optional break. The nearest n-ary operator determines the exact behavior. E.g. `x` may or may not be executed in `[break? x] y`; this is much like `[[+] + x] y`.
- `...` marks a loop; it is equivalent to `while(true)`.
- `..?` marks a loop and at the same time an optional break.

4.7 Scripts and Calls

A SubScript implementation will translate each script into a method that has return type `Script[T]` where `T` is the type of the script's result value (see below). 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.

The body of the example script `test` in section 4.1 contains a call to script `hello`. This is much like a method call.

A script expression may also contain value terms such as variables, literals and Scala code between `()` or `{}`. If such a term is of type `Unit` then it is assumed to be in a tiny code fragment; if it is of type `Script[T]` then it is a script call; else there should be an implicit conversion to a `Script[T]`.

ACP processes supports process communications as atomic actions that are shared by two or more parties. In SubScript this has been generalized to shared scripts that are called by multiple parties. E.g.

```
send, receive = {! println("Communication") !}
```

Synchronous calls to `send` and `receive` that do not exclude one another, may result in the execution of the shared script body. This is also a generalization of normal script calls; the latter may be considered to be special cases of communication with only one party involved.

4.8 Script Lambdas

For Scala value expressions there is a new kind of term: parameterless script lambdas (AKA closures). These appear as script expressions placed between rectangular brackets, such as `[[a b+c] d]`. These values of type `Script[T]` for some type T.

The Scala way of defining parameterized lambda expressions applies as well, essentially giving parameterized script lambdas, e.g., `(i:Int) => [{:print(i):}]`.

[9] These operators start with a percent sign; they are members of a larger family of operators that can suspend and resume operands. These operators are not meant to be memorized; rather they may be encapsulated in higher level scripts with descriptive names.

4.9 Result Values

Code fragments and scripts 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 caret as a postfix operator indicates that the script's result value is set from its operand. The second script has only one operand that is a script call or code fragment; in such cases the caret may be omitted.

The notation `^5` is shorthand for `{:5:}^`.[10]

A double caret is useful for operands that appear in loops, as in `..? x^^`. The result of these zero or more `x`'s is a list; on each success of an `x`, its result value is copied to the list at the position corresponding to the position of that `x` in the loop.

Double carets that immediately followed by integers creates a result tuple. E.g.

```
s = {:1:}^^1 {!"str"!}^^2
```

will produce a tuple `(1,"str")`, of type `(Int,String)`.

`[x]^` is shorthand for `([x])^`, and likewise for double carets etc. This meaning is as follows: the parentheses enclose a Scala value, which is a script lambda having `x` as body. That has a `Script` type, which implies that the whole is a script call. But it is not entirely the same as `x`. Such a construct is useful for more complex result structures such as lists of tuples. E.g.,

```
s = ..? [x^^1 y^^2]^^
```

results in a `List[(X,Y)]`, where `X` and `Y` are the result types of `x` and `y`.

Apart from having success, a script may terminate in failure; that will often be due to an exception thrown from within a code fragment. The exception should be available as an alternative kind of result, similar to what can happen in *future* constructs used in functional reactive programming. Like in futures, a normal result is packed in a `Success` container, and an exception is packed in a `Failure` container.

5 The SubScript Virtual Machine

SubScript implementations have a Virtual Machine that executes scripts by internally doing graph manipulation.

5.1 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,

[10] `5^` is also valid syntax; this requires an implicit conversion to be in scope that turns the number into a script call

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

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

The code generated for the script closure calls library methods that build so-called "template trees", representing the static structure of the invoked scripts. Based on these template trees the script executor maintains a so called call graph. This is a generalization of a regular call stack. It is an acyclic graph; under its root node other nodes will be added and removed according to the template tree as the program is executed. These nodes represent process expression constructs, such as script calls, n-ary operators and code fragments. Recursive script calls lead to repeated occurrences, like in a call stack.

Each type of node has its own typical kind of life cycle. The executor maintains a prioritized queue of messages that direct the state transitions along these life cycles.

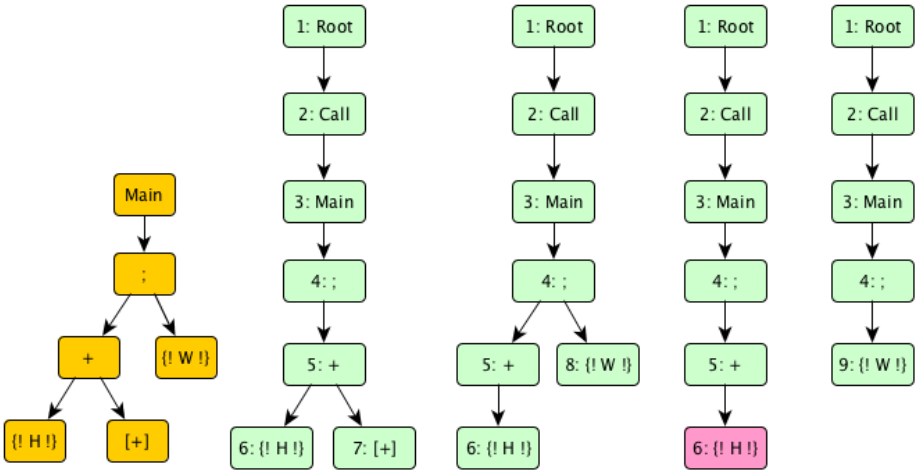
For instance, consider the following process which prints optionally "Hello", and then "world!":

```
Main = [{!print("H")!} + [+]] {!print("W")!}
```

`[+]` corresponds with 1 in ACP. Given the equivalence $(x + 1) \cdot y = x \cdot y + y$), this process should behave much like

```
Main = {!print("H")!} {!print("W")!} + {!print("W")!}
```

The following figure gives the template tree (in yellow) and 4 typical stages of the call graph (in green and red):



In the first depicted call graph, the left hand side of the semicolon has been activated. Node 7 now succeeds. Its success is propagated upwards, through node 5 until node 4.

This node does not react immediately; that has to wait until there is no more graph management to be done at its descendant nodes. Indeed, node 7 deactivates. Then node

4 takes action. Because of the received success, this sequential operator looks up in the template whether there would be a next operand to activate; indeed there is one, which will become node 8.

At that point there is no more call graph management to do. Then the code fragment of node 6 is executed (depicted in red). At the same time a conceptual atomic action happens; a message about this is propagated upwards in the call graph. The sequential node 4 reacts to this by excluding all its other operands; in this respect it acts the same as a `plus` node. Thus the branch with node 8 is deactivated as soon as node 6 starts executing.

After the execution, node 6 will succeed, and then again node 4 activates the next operand from the template, this time as node 9. Node 6 is deactivated, as is its parent node 5. The code fragment of node 9 executes. Then a success follows which propagates upwards to the root. Finally node 9 and its ancestors are deactivated.

Call graph management has a higher priority than executing code for atomic actions. Graph operations below a unary or n-ary operator has a higher priority than such operations at such an operator. This is achieved by collecting messages arriving at such operators in so called `Continuation` messages. This way the response at the n-ary operator can take into account all messages that have arrived.

The message types, in descending priority order, are:

- `Activation` - a node is to be added to the call graph, according to the template tree. This may also involve executing native code for annotations, process parameter evaluation, if- and while conditions, etc.
- `CFActivated` - a code fragment has been activated
- `AAHappened` - an atomic action has happened
- `Break` - a break has been encountered; a flag indicates whether it is optional
- `Success` - a success has been encountered
- `Exclude, Suspend, Resume` - atomic actions in descendants must be excluded, suspended or resumed
- `Deactivation` - a node is to be removed from the graph
- `Continuation` - collected messages for an operator node
- `CFExecutionFinished` - the execution of a code fragment has finished
- `CFToBeExecuted` - a code fragment is to be executed in the main thread

Messages `Exclude`, `Resume` and `Suspend` are propagated downwards in the call graph. Messages `AAHappened`, `CAActivated`, `CFActivated` are propagated upwards in the graph; they may have effects at the nodes that they pass by.

E.g., when a `AAHappened` message arrives from a child node at a `+` or `;` node, `Exclude` messages for the sibling nodes are inserted in the message queue.

`Break` and `Success` are also propagated upwards, up to n-ary nodes. Such nodes have different ways to handle `Success` messages that arrived from their child nodes; often these result in sending new `Success` themselves.

Process communication involves multiple callers that call a single shared script. This binds branches of the call graph together; without communication the call graph would be a tree.

5.2 Implementation

The first SubScript implementation consisted only of a SubScript Virtual Machine: a library written in Scala, called from code with plain Scala syntax. The VM had 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 this approach 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 important for clarity.

Therefore we made a special branch of the Scala compiler that translated the genuine SubScript syntax to the library calls. This took about 2000 lines of Scala code, mainly in the scanner, the parser and the typer. As of 2015 by a Parboiled[14] based preprocessor parses SubScript sources and generates Scala with some dedicated macro calls therein; the standard Scala compiler is thereafter invoked. This approach is leaner and more maintainable; however it leads to inconvenient compile error messages, and it is not suited for IDE integration.

6 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 use a ternary version of the dataflow operator:

```
exit = exitCommand
       @gui: {! confirmExit !} ~~(b:Boolean  )~~> while !b
                               +~/~(e:Exception)~~> {:println(e);}
       ...
```

In case `confirmExit` would throw an exception, the code fragment would end in failure and its result would be a `Failure` wrapper containing the exception. Next, because of the failure, the arrow part with the slash would be followed, so that the exception is printed. The periods on the last line enforce that the script is a loop, even in case `while` has not been reached.

In general the dataflow operator can become analogous to a combination of match statements and exception handers. E.g., the dataflow on the left is syntactic sugar for a lower level dataflow on the right:

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

So it comes down to the meaning of `x~~>y+~/~>z`. In such a dataflow, `y` and `z` must be partial scripts, i.e. partial functions that return a `Script[T]` for some type `T`. The dataflow starts with `x`. When `x` has success, `y` is activated with `x`'s normal result value passed as actual parameter. When `x` terminates as a failure, `z` is activated with `x`'s resulting exception passed as actual parameter.

`x~~>y` is similar, except for that it ends in failure when `x` ends in failure.

`x~/~>z` is also similar, except for that it succeeds when `x` succeeds.

6.1 Example: Twitter Search

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)

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

The slash and the iterator 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 event 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.

A ternary dataflow operator directs the search result (of `searchTweets`) to the either `updateView` or `setErrorMsg`..

It is possible to create an implicit script that converts a future into an appropriate script. If such an implicit script were in scope, we may replace the threaded code fragment `{*searchTweets*}` by the future `fSearch`.

7 Related Work

Since the predecessor paper [5] we have improved the features for result values and dataflow.[11] The dataflow support now also covers pattern matching and exception handling. This improves the cooperation with futures and actors.

The predecessor paper contains an overview of other languages that show some resemblance to this work. Grammar notation formalisms are most related, as these have similar support for sequences and choices. SubScript result values were inspired by YACC[10] and by the parser combinator library FastParse[12].

SubScript has a delayed task execution. This also occurs in futures and the async idiom, known from functional reactive programming. Futures may terminate successfully or in a failure, which comparable to SubScript scripts; however they lack alternative compositions and a 1 element. In a way SubScript adds the expressiveness of grammar formalisms to the concurrency domain.

Other related approaches are Reactive-C [3] and its follow up SugarCubes [4]. These two have a similar execution mechanism with call graphs; yet they are not process algebra implementations since also they lack alternative compositions and a 1 element.

There are some papers that apply process algebra as a theoretical underpinning to actors: [1], [7] use Pi-calculus, and [13] applies ACP.

8 Conclusion

SubScript offers constructs from the Algebra of Communicating Processes that supports a declarative programming style. This is useful for GUI controllers, text parsers and probably other areas.

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.

[11] A useful definition for `[x]^` (see section 4.9) triggered several syntax changes. E.g. rectangular brackets replaced parentheses to delimit process expressions. Script lambda's are now also written between rectangular brackets. Script terms may now have the form `(s)` or `{s}`, with `s` a Scala value; such terms are method calls or script calls, possibly after implicit conversion. Normal code fragments had the form `{s}`; this became `{!s!}`.

[12] http://lihaoyi.github.io/fastparse/

The performance is typically in the order of 10,000 actions per second on current mainstream personal computers. For most GUI controllers this speed is acceptable; for text processing that would depend on the input size.

SubScript is an open source project[13]. It is currently implemented as a branch of the regular Scala compiler, bundled with a virtual machine and libraries for interfacing with Akka actors and Swing GUIs.

9 Acknowledgement

We thank the referees and especially the shepherd for their useful suggestions and other comments.

References

[1] Agha, G., Thati, P.: An algebraic theory of actors and its application to a simple object-based language. In: In Ole-Johan DahlÕs Festschrift, volume 2635 of LNCS. pp. 26–57. Springer (2004)
[2] Baeten, J.C.M.: A brief history of process algebra. Theor. Comput. Sci. 335, 131–146 (May 2005)
[3] Boussinot, F.: Reactive c: An extension of c to program reactive systems (1991)
[4] Boussinot, F., Susini, J.F.: The sugarcubes tool box. In: Nets of Reactive Processes Implementation
[5] van Delft, A.: Dataflow constructs for a language extension based on the algebra of communicating processes. In: Proc. 4th Workshop on Scala. SCALA '13, ACM (2013)
[6] van Delft, A.: Some new directions for ACP research. CoRR abs/1504.03719 (2015), http://arxiv.org/abs/1504.03719
[7] Gaspari, M., Zavattaro, G.: An algebra of actors. In: Ciancarini, P., Fantechi, A., Gorrieri, R. (eds.) FMOODS. IFIP Conference Proceedings, vol. 139. Kluwer (1999)
[8] Goeman, H.: Towards a theory of (self) applicative communicating processes: A short note. Inf. Process. Lett. 34(3), 139–142 (1990)
[9] Hoare, C.: Communicating sequential processes. ACM Computing Surveys 7(1), 80–112 (1985)
[10] Johnson, S.: Yacc: Yet another compiler- compiler. Tech. rep., Bell Laboratories (1979)
[11] Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1982)
[12] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part i. Information and Computation 100 (1989)
[13] Wang, Y.: Fully abstract game semantics for actors. CoRR abs/1403.6563 (2014)
[14] Wills, P.: No more regular expressions. Scala Exchange, Skills Matter, London (2014)

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