Dataflow Constructs for a Language Extension Based on the Algebra of Communicating Processes

André van Delft
andre dot vandelft at gmail dot com

ABSTRACT

The dataflow programming paradigm addresses how data flows inside programs. Program components, that are often concurrently active, send data to one another; this eases software composition. Mainstream programming languages do not support the paradigm well because of their deterministic and sequential nature.
A language that focuses on concurrency is better suited to incorporate concepts from the dataflow paradigm. SubScript is an extension to the Scala programming language with constructs from the Algebra of Communicating Processes, targeted at event-driven and concurrent programming. Like ACP, SubScript focuses on program behavior; support for data was through local variables and parameters. However, the ACP background enabled SubScript to deal with the challenges of the dataflow paradigm. This is achieved through several new features.
1. A process may have a result value, like a method
2. A process result value may be passed on to another process that starts subsequently. This helps getting rid of variables, e.g., in GUI controller specifications.
3. Output actions from a process may be piped to a parallel process, yielding a similar expressiveness as pipes in Unix command shell language.
4. Actors written in Scala often need to keep track of a state, and their program text poorly expresses the conceptual control flow. When such actors are written in SubScript, incoming data may be treated as events that may appear anywhere in the specification, just like in SubScript GUI specifications.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Dataflow languages

General Terms

Languages, Theory

Keywords

Algebra of Communicating Processes, dataflow, concurrency, nondeterminism, GUI programming, actors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Scala '13, Montpellier, France
Copyright 2013 ACM 978-1-4503-2064-1 ...$15.00.

1. INTRODUCTION

Dataflow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations[1]. Program components that are often concurrently active send data to one another. This is done in Unix shell languages: small single-purpose tools are easily glued together using the pipeline symbol: `|`. This ease of use has made Unix command shell pipes attractive and popular. Pipes were possible thanks to the concurrency support by Unix. This is largely lacking in mainstream programming languages, which are deterministic and sequential.
A programming language that is focused on concurrency is better equipped to support the dataflow paradigm. SubScript [van Delft()] is an extension to Scala [Odersky et al.(2008)Odersky, Spoon, and Venners][2] with constructs from the Algebra of Communicating Processes (ACP) [Baeten(2005)]. It is similar to grammar specification languages such as YACC [Johnson(1979)]; a main difference is that SubScript also has elementary support for concurrency. It is simple to glue processes to one another, but dataflow constructs were lacking. The formalism focuses on program behavior, yielding concise specifications. Support for data was initially through local variables and parameters for process refinements. There much of the conciseness could be lost.
This paper offers four new kinds of dataflow support in SubScript:
1. A process refinement may have a result value, like a method.
2. One-time flow between processes is supported by allowing for result values flowing to subsequent processes. This is depicted by a long arrow: `==>`.
3. Multiple-time flow is supported by communication through pipes; internally the processes repeatedly perform write and read actions over these pipes. Pipes are depicted as parallel operators followed by a long arrow, such as `&==>`.
4. An actor [Carl Hewitt(1973)] framework such as Akka[3] has support for actor topologies that are much more general than pipes. Actors may be distributed over a network. Callback methods for incoming messages describe Akka actor behavior; this makes control flow hard to understand. Using SubScript the behavior may be described as a process without the need for callbacks. Actors in plain Scala use partial functions to specify the expected messages; in SubScript a similar "partial script" is applied.
The applied arrow symbols give a visual indication of dataflow. As in mathematics, notation is not just a detail; it matters for clarity.
The next chapters discuss ACP, SubScript and its implementation. Thereafter use cases of text parsers, GUI controllers, pipes and actors in SubScript code show the new data flow support.

[1] Formulation taken from Wikipedia on "Dataflow programming"
[2] In principle the extension may apply also to other languages such as C, C#, Java and JavaScript
[3] See http://doc.akka.io/docs/akka/snapshot/scala/actors.html

2. THE ALGEBRA OF COMMUNICATING PROCESSES

The Algebra of Communicating Processes (ACP)[Baeten(2005)] 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[4]. More so than the other seminal process calculi (CCS [Milner(1982)] and CSP [Hoare(1985)]), 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 δ and ϵ behave much like the 0 and 1 that are usually neutral elements for addition and multiplication in algebra:

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

There is no axiom for $x \cdot \delta$. $x + \epsilon$ means: *optionally x*. This is illustrated by rewriting $(x + \epsilon) \cdot y$ using the given axioms:

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

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

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

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

- $x \llfloor y$ - "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: $(x + y)$, $a \cdot x$, ϵ, δ. E.g. the axioms for the left merge operator are:

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

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 [Goeman(1990)]. Shortly thereafter, Robin Milner et al developed Pi-calculus [Milner et al.(1989)Milner, Parrow, and Walker], which also combines the two theories.

3. SUBSCRIPT

SubScript mainly offers a new 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.

Esthetically, ACP processes are preferably notated with the mathematical expression syntax. However, ACP symbols $\cdot$, $\|$, δ, ϵ 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.

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 interoperate with genuine Scala code:

- Scala code may call scripts as if these were a special kind of methods. An extra parameter would apply for such calls: the process executor, which may be tailored for the type of application. After the call from the native code ends, the executor may provide information on the execution, e.g. on whether the script ended successfully or as deadlock (δ).

- Any fragment of Scala code placed between clear markers such as braces may serve as an operand in a process expression. The start and end of such fragments will correspond with atomic actions in ACP. This way the code fragments may overlap, which is useful when they are run in separate threads, or when they denote actions that take some simulation time in a discrete event simulation context.

Scripts are usually defined together in a section, e.g.,

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

def testBridge = subscript.DSL.\_execute(test)
```

From here on the header `script..` is mostly omitted for brevity. The DSL method `_execute` this calls the Scala method generated for the script `test` with a fresh CommonExecutor as a parameter; this is the default executor type. Other types of executors could be more suited for specific application domains, such as discrete event simulations and multicore 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,". The order is up to the executor; the CommonExecutor will deterministically apply a left-to-right preference.
In general the atomic actions in concurrent processes are shuffle merged, like one can shuffle card decks.
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 type parameters and data parameters; each parameter may be named or implicit. Variable length parameters and even script currying are possible.

3.1 Script parameters

Next to value parameters, scripts may have *output* parameters. These are prefixed by a question mark in both the declaration header and in the actual call. E.g., with a declaration
`receive(?c:Char)` a call would look like `receive(?d)`.
A double question mark in the declaration marks a *constrained* parameter, e.g., `key(??c:Char)`. Then at the actual call four kinds of parameters may be provided:

- `key(?d)` - any character
- `key('x')` - the character 'x'
- `key(?d ?if(_.isUpper))` - any uppercase character
- `key(??d)` - `d` is also a *constrained* parameter of the script that contains this call, and the parameter kind at the actual outer call applies as well to the inner call.

A slot for an actual output parameter is often taken by a local variable, as in `var d:Char key(?d)`. Alternatively parameter lists do not need parentheses; a starting comma is enough: `var d:Char key,?d`. This leads to a useful shorthand notation `key,?d:Char`. When the script `key` is declared as implicit, the actual call may even be `?d:Char`.

3.2 Execution modes

SubScript supports various execution modes for code fragments. E.g., script expressions may contain mixes of input actions and output actions. Input actions would correspond to code fragments executed by event listeners. Output actions run by default in the main thread, but they may be forced to run in the GUI thread, a database thread or in its own thread. Script expressions may be annotated, allowing for a flexible set of execution modes supported by an imported library and the actual executor. E.g., `@processor=2: {*someCode*}` could specify that code fragment is scheduled for processor 2. Likewise a code fragment may be associated with a specific start time or duration; time may then be either real time or a simulation time.

3.3 Logic extensions to ACP

As ACP is a kind of extension of Boolean Algebra, processes have logic properties: they may fail (ending in δ) or succeed (ending with ϵ or with an atomic action). This suggests more kinds of parallelism are possible. In ACP parallelism all operands should succeed for the operator to succeed; in SubScript this is called and-parallelism. Likewise or-parallelism requires only one operand to succeed. For both kinds there are "strong" flavors:

- in strong and-parallelism, when one operand fails all others are forced to stop
- in strong or-parallelism, when one operand ends successfully all others are forced to stop

This has an analogy with boolean operators in C-like language: `&&` and `||` do not evaluate the second operand if the evaluation of the first one is decisive. Therefore familiar symbols `&`, `|`, `&&` and `||` depict these four flavors of parallelism in SubScript. Other logic operators are for instance process negation (inverting the logic result of a process) and failure continuation (a sequence that continues to the right as long as operands fail).

3.4 Communication

Traditional ACP process communication is binary (it involves 2 communicating parties), and it yields an atomic action. These restrictions may be lifted: communication may be n-ary, and yield any process expression. A special case is then unary communication, which is an other way to describe process refinement. Multiparty communication may likewise be viewed as "multi-calls": multiple callers (e.g. a send action and a receive action) perform together a single shared process.
Other forms of communication are now available at little cost:
An asynchronous send action is simply a process launch of a normal send action. Some syntactic sugar eases the specification of communication over channels, as in:

```
c<-->(?i: Int) = {}
test           = c<-3 & c->?j: Int
```

Names of process refinements may for this purpose end in an arrow <- and ->. The part of the name before the arrow may be empty. Thicker arrows <= and => offer similar kind of channel communication that is subjected to a network topology, as laid out later in this paper.

3.5 Deterministic control and Iteration

SubScript has `if-else` and `switch` constructs as could be expected. Five operand types support iterations and breaking:

`while`	marks a loop and a conditional mandatory break
`...`	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

Note that these are operands; they often belong to a sequential operator, but the iterators may as well relate to another n-ary operator such as a parallel one.

4. CALL GRAPH SEMANTICS

A SubScript program is executed by a common or specific executor. The semantics therefore depends on the applied executor. A common should preferably be a reference for other kinds of executors. However, no exact specification for a common executor is available at the time of writing this article. Yet we can present informally how it should operate.

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

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

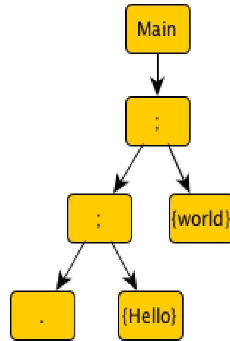

Figure 1: Template Tree

A process being called from the base language implies that its template tree is handed to a process executor. The latter starts building a so called call graph, an acyclic graph with a single root node. Below this root node there is a "process call node", with in turn parents a "callee node" for the executed process. Under that, other nodes will be added and removed according to the template tree as the program evolves; these nodes represent the process expression constructs, such as n-ary operators and code fragments. This is done by handling simple messages of various kinds.
Call graph management precedes over executing code for atomic actions. Graph operations below a unary or n-ary operator precede over the operation at such an operator. This is achieved by collecting messages arriving at such operators in so called `Continuation` messages. This way the response by at the n-ary operator can take into account all messages that have arrived.
Some message types in descending priority order:

- `AAActivated, AAStarted, AAEnded` - an atomic action has been activated, started or ended
- `Break` - a break or optional break has been encountered
- `Success` - a success has been encountered
- `Exclude, Suspend, Resume` - atomic actions in descendants must be excluded, suspended or resumed
- `Activation` - a node is 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. Moreover, one or more messages may be inserted in the message queue.
- `Deactivation` - a node is removed from the graph
- `Continuation` - Collected messages for an operator node
- `AAExecutionFinished` - a code executor reports that the code related to an atomic action has finished
- `AAToBeExecuted` - Atomic Action to be executed in the main thread

Messages `Exclude`, `Resume` and `Suspend` are propagated downwards in the call graph. Messages `AAStarted`, `AAEnded`, `CAActivated`, `AAActivated`, `Break` and `Success` are propagated upwards in the graph; the latter two stop at n-ary operator nodes. They may cause something to happen; e.g., when an `AAStarted` message arrives a child node at + and `;`, `Exclude` messages for the siblings are inserted.

Many language constructs behave quite straightforward. E.g., upon activation, the deadlock process (symbolized by `(-)`) inserts a `Deactivation` message. The empty process (symbolized by code(+)) inserts a `Success` and a `Deactivation` message. The neutral process determines whether its n-ary operator ancestor is and-like: in that case it acts as `(+)`, else as `(-)`. When an iterator is activated (`..` `...` `while`) it first sets an iteration flag at its n-ary ancestor node, if any. Then it may execute some test (`while`), insert a `Break` message; from then the iterator acts as the neutral process (which has symbol `(+-)`).

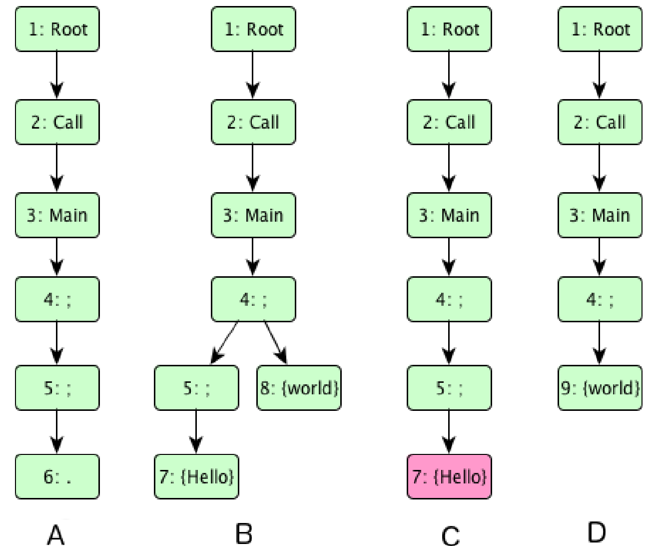

Figure 2: Call Graphs

Figure 2 shows the evolution of the call graph. Nodes 1 and 2 are the root and an anchor for the executed process, node 3. This activates node 4: the outer sequential operator. Activating any n-ary operator activates its leftmost operand, and inserts a `Continuation` message. Thus nodes 5 and 6 are activated (see A).
The latter inserts `Break`, `Success` and `Deactivate` messages. These three arrive at sequential node 5, where they are added to the `Continuation`; the `Deactivation` also removes node 6. The `Continuation` at node 5 activates the next operand (node 7, Hello), because of the `Success`. It also sends an `Success` onward to node 4, because of the optional `Break`. There it is added to the `Continuation`. Handling the `Success`-holding `Continuation` at node 4 activates the next operand (node 8, `World`). During the activation of nodes 7 and 8, `AAActivated` and `AAToBeExecuted` messages are inserted (see B).
Now the `AAToBeExecuted` for node 7 is handled; it precedes over the one for node 8 because it had been inserted earlier. The code executor for node 7 executes the code and inserts `AAStarted`, `AAEnded`, `Success` and `Deactivation` messages. The message `AAStarted` has no effect at node 5. There another `AAStarted` is inserted propagating to node 4. Handling this one causes an `Exclude` message for node 8 to be inserted. Handling that message inserts a deactivation for node 8 (see C).
Then another `Success`-holding `Continuation` at node 5 is handled; this leads to such a `Continuation` at node 4, which activates the node 9, again for `World` (see D). After that has been executed, all nodes are deactivated and the process execution ends.
An Unsure code fragment would be executed first; if it succeeds, messages are inserted like for a normal code fragment. If it fails, a Deactivation is inserted. If it got an undetermined state then a `AAToBeReexecuted` would be inserted.
For a code fragment that runs in its own thread or in the GUI thread, an `AAStarted` message is inserted, and thereafter the

code fragment is executed asynchronously; at the end thereof an AAExecutionFinished message is inserted. This is picked up in the message handling loop, and then, normally, AAEnded, Success and Deactivation messages are inserted.
An event handling code executor works in general asynchronously. Upon an event notification it executes the code, and then inserts an AAExecutionFinished message. Handling the latter inserts an AAStarted message and the three others.
Note that handling an AAExecutionFinished may result in only a Deactivation message, when the atomic action had been excluded meanwhile, e.g. because of another atomic action in another branch of an disrupt operator.

5. 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 2000 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.

5.1 Compilation

The already presented script

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

is translated during compilation into the following DSL code:

```
def Main = _script('Main) {
 _seq(
  _seq(_optionalBreak,
   _normal{(here:N_code_normal)=>print("Hello ")}),
  _normal{ (here:N_code_normal)=>print("world!")})
}
```

5.2 VM

The DSL _execute method invokes a method named run in the SubScript VM. A bit simplified version is like:

```
def run = {initializeWork; while (workToDo){}}
def initializeWork = activateFrom(anchorNode,
                          anchorNode.t_callee)
def workToDo(): Boolean = {
  if (callGraphMessageCount > 0)
    handle(dequeueCallGraphMessage)
  else if (!rootNode.children.isEmpty) {
    synchronized {
      if (callGraphMessageCount==0)
        synchronized {
          wait() // for an event to happen
  } }   }
  else return false
  return true
}
```

The SubScript VM may have to collaborate with frameworks that do a main loop by themselves. Then in principle two solutions are possible:

- SubScript's run method is called in a special thread, and this runs in synchronization with the framework's main loop
- SubScript's run method is not called; instead the initializeWork is called and thereafter the framework loop calls regularly the workToDo method

Thread synchronization is scarcely needed in the source code; synchronized code is mainly needed for accessing the message queue. GUI example applications run responsively. The interaction with the underlying GUI library does not involve polling; there is no CPU time spent when waiting for input. The interaction is pretty much the same as a plain Java or Scala would have. A difference is that a widget listener is registered and unregistered each time associated actions such as click are activated and deactivated.
To give an indication of the speed: an implementation of a Java based predecessor language is in production use at an engineering agency for parsing text documents. On a 3GHz Linux PC the throughput per second is 30,000 accepted tokens out of 120,000 expected ones (i.e. the average accepted token had been one among 4 alternatives). This speed suffices in many practical cases: on that platform the overhead is less than 0.1 milliseconds per action; for GUI controllers no slowdown will be noticeable.

6. BEYOND ACP STYLE EXPRESSIONS

SubScript as defined so far is higher level language than Scala, since it allows to express nondeterministic choice and concurrency more concisely. On the other hand, SubScript is very terse in its support for nondeterministic choice and concurrency. It is still hampered in two ways: For the second issue a solution is briefly outlined here.
In Scala one can program a "While" method using anonymous functions; other control structures are equally possible, e.g. for locking. Something similar should be possible using script lambda's (AKA anonymous scripts, analogous to anonymous functions), but preferably with much less parentheses and braces. Suppose script lambda's are written as < expr >.
A little bit of syntactic sugar using the tilde symbol (~) enables programming scripts calls with the script name divided into parts around the parameters. Each name part may or may not be empty. E.g., a progress monitor that during 5 seconds regularly updates a status would be in terse SubScript:

```
progressMonitor = sleep_ms(250) updateStatus ...
               || sleep_ms(5000)
```

With a lambda and the syntactic sugar this could be written as

```
 progressMonitor = during_ms~ 5000
                  ~every_ms~ 250
                  ~do~< updateStatus >~~end
```

using a script with split names (that may come from a library):

```
during_ms~ duration:Int
~every_ms~ interval:Int
~do~ s:script ~end     = sleep_ms(interval) s ...
                      || sleep_ms(duration)
```

The rest of this section focuses on solutions for the first issue: support for data.

6.1 Use Case: Text Parsing

SubScript allows for concise grammar specifications, but getting data out of it involved relatively much coding. E.g., these scripts describe the syntax of numeric expressions such as 1+2*(3+4):

```
  expr   = term   .. "+"
  term   = factor .. "*"
  factor = number + "(" expr ")"
```

A string such as "+" and "*" is here a parameter for an implicit script `parseString`. The low level scripts `parseString` and `number` may be programmed using annotations that registers expectations at a lexical scanner, followed by a code fragment that succeeds when such expectation is met:

```
implicit parseString(s:String)
      = @expStr(s): {?accept?}
number = @expNum   : {?accept?}
```

The specification of `expr`, `term` and `factor` above only checks for grammatical correctness; it does not produce a result. Compare for instance a YACC specification for the same input, that also produces a numeric result:

```
expr   : expr PLUS term  { $$ = $1 + $3; }
                |  term  { $$ = $1; } ;
term   : term MUL factor { $$ = $1 * $3; }
                | factor { $$ = $1; } ;
factor : LPAR expr RPAR  { $$ = $2; }
                | NUMBER { $$ = $1; };
```

A SubScript specification with the same functionality would require that after successful termination of the calls to term, and factor an addition or multiplication is performed. These calls need for that purpose to listen to such an event. This could be done using a split-name script `do~s:script~whenDone~f:Unit~end`; in this context a name without letters may be preferable due to its conciseness: `~s:script~~f:Unit~`. The script registers a listener using an annotation:

```
~s:script~~f:Unit~:Int
= @onDeactivate_success{f}: s

expr(?r:Int) = {!r=0!}; var t:Int
             ~< term(?t) >~~r+=t~  .. "+"
term(?r:Int) = {!r=1!}; var t:Int
             ~<factor(?t)>~~r*=t~  .. "*"
```

`{!r=0!}` initializes the result value of script `expr`. The braces with exclamation marks state that this code fragment is "tiny", i.e. immediately to be executed upon activation, and it does not correspond to conceptual atomic actions. The semicolon following the initializer ensures that the initializer is not part of the loop created by the two periods to the right.

The `factor` script does an implicit call to an implicit parameterized `num` script:

```
      factor(?n:Int) = ?n + "(" expr,?n ")"
implicit num(??n:Int) = @expNum(_n):{?accept?}
```

The `num` script has two question marks prefixing its parameter, so that an actual call may supply an output parameter, but also a specific "matching" value. The `expNum` method will take care of that; it needs to access more information on the parameter (e.g. is it called as an output parameter or with a forcing value). This information is available in _n, i.e. a holder for the official formal parameter `n`, marked by an underscore prefix:
In the specification above variable declarations distract quite strongly from the described expression syntax. This may improve using the new script result value feature.

```
~s:script[Int]~~f:Int=>Int~: Int
= @onDeactivate_success{$ = f($s)}: s

expr: Int = {!0!}^; ~< term >~~ $ + _ ~^  .. "+"
term: Int = {!1!}^; ~<factor>~~ $ * _ ~^  .. "*"

factor: Int = ?$ + "(" expr^ ")"
```

- a script call and a code fragment may get a caret (^) suffix: this means that the script result value is set to the result value of the call or code fragment.
- in a value expression the pseudo-variable `$` stands for the result of the currently defined script.
- `$name` stands for the last yielded result value of a called script with the given name.

6.2 Use Case: A GUI Controller

GUI controllers may require a sequential composition[5] where the the left hand operand has a result which is a default parameter input for the right hand operand:

```
~[T,U]s:script[T]~~t:T=>script[U]~: U = s; t($s)^
```

A short hand notation for `~<a^>~~<b^>~` is `a==>b`

6.2.1 Example: Mouse Clicks and Key Input

Suppose `click:Point` and `key:Char` are scripts for mouse click events and key events. These could be operated as follows:

```
clickHandler = click ==> handleClick(_); ...
keyHandler   = key   ==> handleKey(  _); ...
```

Note that the semicolons are needed; without these only the right hand side of the arrows would iterate, without the calls to `click` and `key`.

6.2.2 Example: Exiting

GUI Controllers programmed in SubScript often have an exit script that is placed in an or-parallel composition to the main program flow. This way the controller ends its operation as soon as the exit script is ready; the exit script is a sequential loop of each time an exit command followed by a confirmation dialog; the loop ends upon true confirmation:

```
doExit = var sure=false
         exitCommand
         @gui:{sure=areYouSure}
         while(!sure)
\vspace{-3pt}
```

`@gui:` means that the subsequent piece of code must be done in the GUI thread using `SwingUtilities.invokeLater()`. The variable `sure` is declared separately; this way it dominates a large part of the script. Using the result flow arrow this becomes:

```
doExit = exitCommand; @gui:areYouSure ==> while(!_)
```

[5] In fact a special sequence operator would be needed that would shine through for iteration operands. The semicolon is shown here instead, to limit the introduced syntax.

6.3 Use Case: Pipes

SubScript supports pipe communication though variations of parallel operators:

- `&==>` - normal parallelism with dataflow from left to right
- `||==>` - likewise, for strong or-parallelism

Processes that take part in such pipe constructs read and write using special communication actions. These are send and receive scripts of which the names end in an arrow: <= for send and => for receive. The alphanumeric part of the name may be empty so that only an arrow remains.

6.3.1 *Example: Copying a file*

As an example, consider a dataflow program for copying a file. It mainly contains two processes, a reader and a writer, and connects these through a network operator:

```
copier(in: File,
      out: File) = reader(in) &==> writer(out)
```

The reader process is quite simple. It opens a file; then reads bytes from the file and outputs these over the network; when end of file is reached the loop ends and the file is closed. Note that it also transmits the end-of-file (-1) value over the network.

```
reader(f: File)
  = val inStream = new FileInputStream(f);
    val b = inStream.read()  <=b  while (b!=-1);
    inStream.close()
```

The writer is similar: it opens a file; then reads data from the network and, as long as end-of-file is not encountered, it writes these to the output file

```
writer(f: File)
  = val outStream = new FileOutputStream(f);
    =>?b:Int  while (b!=-1)  outStream.write(b);
    outStream.close()
```

Here the `while` construct is the middle in a sequence of 3 operands; this specifies that the sequence is a loop which ends at the while, as soon as its condition evaluates to false. The communication channel for the data also needs to be declared:

```
<==>(b: Int) = {}
```

This implicitly declares two scripts, `<=(b: Int)` and `=>(??b: Int)`, which share a body; this body may happen when a send call is active in a left hand side operand a pipe operator, and a receive call is active in the right hand side. The double question mark indicates that the receiver call has an output parameter for which also a fixed value may be specified.

6.3.2 *Pros and Cons of Pipelines*

Compare dataflow style with traditional style for the copy method:

```
def copy(in: File, out: File): Unit = {
  val  inStream = new FileInputStream (f)
  val outStream = new FileOutputStream(f)
  val eof = false
  while (!eof) {
    val b = inStream.read()
    if (b==-1) eof=true else outStream.write(b)
  }
  inStream.close()
  outStream.close()
}
```

No performance measurements are available yet, but the traditional style program must be an order of magnitude faster. For relatively small files the speed of the dataflow program may be good enough; for larger files it will become slow, with the currently available SubScript Virtual Machine implementation. However, a more advanced SubScript compiler or VM could analyse the program and transform it into something close to the traditional style version.
The strength of the dataflow program is that it untwists two tasks: reading and writing. This way it becomes easier to put some processing between these tasks. Moreover, `reader` and `writer` may well be placed in a library, so that you donÕt have to deal with the file protocol of opening, processing data and closing. Reader and writer communicate over unnamed channels using <= and =>. It is like communicating with standard input and output devices.
This may yield a simple tool set for file handling and data processing; such tools may easily glued together just like is done in Unix shell language.

6.3.3 *Example: Filtering a text file*

It is easy to create a file filter using the given reader and writer. E.g., the following filter would eat away carriage-return characters:

```
filter(in: File,
      out: File) = reader(in) &==> charFilter
                              &==> writer(out)
charFilter = =>?b:Int
             if(b!='\r') <=b
             while(b!=-1)
```

6.3.4 *Example: Encoding and Decoding a text file*

The dataflow programming style combines well with specifying some syntax in the flows. Consider the task to encode and decode text files with run length compression. In the encoded file

- a backslash character and a digit are replaced by an escape sequence starting with a backslash
- a run of two or more times the same character is replaced by a single occurrence of that character followed by an indicator of the run length; the indicator is a string representation in reverse order.

Network pipes with file readers and writers at both ends will do the encoding and decoding:

```
fileEncoder = reader,in&==>encoder&==>writer,out
fileDecoder = reader,in&==>decoder&==>writer,out
```

For the encoder and decoder we need a script `lowPriority`, which comes down to an action that only happens when there is nothing else to do. The `encoder` and `decoder` scripts may reflect the grammar of the unencoded file and the encoded file. Loosely formulated:

```
unencodedFile = ..; anyChar; .. sameChar
  encodedFile = ..; . '\\'; anyChar; .. digit
```

Here `sameChar` denotes the same character as previously seen as `anyChar`. The encoder has a loop:

- read a value from the network
- read zero or more times the same value again (and counting the occurrences); specify a "forcing" parameter
- the `lowPriority` action so that the loop is only exited when no more same value arrive
- write the escape character over the pipe, if necessary
- write the value over the pipe
- write the run length if it exceeds 1
- the loop ends after processing the end-of-file value (-1)

```
encoder = =>?c: Int; var n=1;
          ..=>c {n+=1}; // c is "forcing"
          lowPriority;
          if (c=='\\'||c.toChar.isDigit) <='\\';
          <=c;
          if (n>1)
             (for(d<-n.toString.reverse) <=d);
          while (c != -1)
```

The decoder also has a loop:

- optionally read the escape character (\) from the pipe
- read a character from the pipe; a preceding `lowPriority` action makes sure this does not grab the escape character
- optionally read a sequence of digits; use these to reconstruct the encoded run length
- the `lowPriority` action, so that the digits must have been read when available
- write the character once or more times, depending on whether a run length was give
- the loop ends after processing the end-of-file value (-1)

```
decoder = .=>'\\';
          lowPriority;
          =>?c: Int;
          var n = 0;
          var p10 = 1; // current power of 10
          ..=>?c ?if _.toChar.isDigit
             {n+=c.toChar.asDigit*p10; p10*=10};
          lowPriority;
          times(max(n,1)) <=c;
          while (c!=-1)
```

The test `?if _.toChar.isDigit` is a requirement for the receive action `=>?c`.

6.4 Use Case: SubScript Actors

Programming the control flow of actors in Java or Scala is relatively hard, just like with GUIs. In both cases events arrive as calls to listeners; these listeners then perform some actions. After this call-back-call the next one will occur at some point. Both GUI applications and actors may change a state and change the set of events that they listen to
With SubScript the control flow may be inverted; scripts treat events and internal actions in an equal way. State is largely maintained implicitly in these scripts. We describe how SubScript does this on top of Akka actors. The full power of the Akka framework remains available because the same things happen under the hood as in plain Scala versions of the actors:

- There are still partial functions listening to incoming data
- Unexpected messages will be result in a call to "unhandled", just like what is normally done in Akka. (SubScript channels are less forgiving, just like Scala Actors were in the past.)

6.4.1 *Example: Akka's ExampleActor*

The `ExampleActor` from the Akka documentation[6] contains the following method:

```
def receive = {
  case Request   (r)  => sender ! calculate(r)
  case Shutdown       => context.stop(self)
  case Dangerous (r)  => a.tell(Work(r),sender)
  case OtherJob  (r)  => a!JobRequest(r,sender)
  case JobReply(r,s)  => s ! r
}
```

A bit hidden in this specification is the fact that processing stops when a Shutdown message arrives. A SubScript variant would not have the `receive` callback method; instead there will be a `live` script, like in SubScript GUI controllers:

```
live = .. <<
  case Request  (r)   => {sender ! calculate(r)}
  case Dangerous(r)   => {a.tell(Work(r),sender)}
  case OtherJob (r)   => {a!JobRequest(r,sender)}
  case JobReply(r,s)  => {s!r}
  >> ;
  <<Shutdown>>
```

This `live` script is willing to receive and process zero or more messages until a `ShutDown` message. After the live script ends `context.stop` will be called, somewhere under the hood.

A «`...`» section contains a so called "partial script". This is much like a Scala partial function; the main difference being that the refinement bodies are scripts.
To the right of the arrows the value `sender` denotes the actor that sent the value specified at the left hand side of the arrow. This is a bit different from the `sender` in class Actor, which denotes the sender of the last received message. That definition could be misleading in SubScript actors when multiple «`...`» sections are concurrently active.
Two rules enable the concise notation for «`Shutdown`»:

- if there is only 1 case the `case` keyword may be omitted
- if there is nothing to do in the body this may be left out together with the arrow

6.4.2 *Example: A Finite State Machine*

The Akka documentation contains an example of the support for Finite State Machines (FSM): *Consider an actor which shall receive and queue messages while they arrive in a burst and send them on after the burst ended or a flush request is received.*
The actor receives the following messages:

```
case class SetTarget(ref: ActorRef)
case class Queue(obj: Any)
case object Flush
```

The `SetTarget` message should be the first received message. As soon as the first `Queue` message arrives a timeout period starts; on this timeout, or earlier when a `Flush` message arrives, the queue is flushed, by sending the following message to the target actor.

```
case class Batch(obj: Seq[Any])
```

For unexpected incoming messages (e.g. an extra `SetTarget` message) warnings should be logged.

[6] See http://doc.akka.io/api/akka/2.1.2/index.html#akka.actor.Actor

The plain Scala solution from the Akka documentation is:

```
sealed trait State
case object Idle   extends State
case object Active extends State

sealed trait Data
case object Uninitialized extends Data
case class Todo(target: ActorRef,
                 queue: Seq[Any]) extends Data

class Buncher extends Actor with FSM[State,Data]
{
  startWith(Idle, Uninitialized)

  when(Idle) {
    case Event(SetTarget(ref), Uninitialized)
      => stay using Todo(ref, Vector.empty)}

  onTransition {
    case Active -> Idle
      => stateData match {
             case Todo(ref,q) => ref ! Batch(q)
  } }

  when(Active, stateTimeout = 1 second) {
    case Event(Flush | StateTimeout, t: Todo)
      => goto(Idle)
             using t.copy(queue = Vector.empty)}

  whenUnhandled {// common code for both states
    case Event(Queue(obj), t @ Todo(_, v))
      => goto(Active) using t.copy(queue=v:+obj)

    case Event(e, s) => log.warning(
         "unhandled request {} in state {}/{}",
                                 e, stateName, s)
     stay
  }
  initialize
}
```

The SubScript version is shorter:

```
class SubScriptBuncher extends SubScriptActor {
  val timer = new ScriptTimer
  var target: ActorRef = _
  var q: Seq[Any] = _ // q shorthand for queue

 script..
  live = <<SetTarget(ref) => {target=ref;
                              q=Vector.empty}>>
       ( .. <<Queue(obj) => {q+=obj}>>
            if (pass==0) timer.start
       ; <<Flush>> + timer.timeout(1 second)
       ; {target!Batch(q); q=Vector.empty;
          timer.stop}
       ; ...
       )
 def unhandled = {
   case e=>log.warning(
            "received unhandled request {}", e)
} }
```

Here `timer` is an object with a start method and a script that handles a timeout event after a given time has passed by.
`pass` is a loop counter; the condition `pass==0` makes sure the timer gets started only once in each burst.
Note that the logged warning message does not give state information, unlike the plain Scala solution.

6.4.3 Example: Parallel Computation

An actor typically performs a task upon request and sends the results back, or to another actor. To exploit parallelism for quick calculation, the actor may split the received task and delegate the parts to newly created other actors. After all delegates have returned their results, the aggregate result is available and the delegating actor is ready to send it on. The following code does this in conventional Akka style [7]:

```
var initializationReady = false
var activeActors = 0
var sum: Double = 0

def receive = {
 case context: Context =>
   sum = 0 //reset the instance variables
   activeActors = 0
   for(task <- context.tasks) {
     val actor = actorOf[Delegate].start
     actor ! DoTask(task)
     activeActors += 1
   }
   initializationReady = true

 case delegateResult : Double =>
   sum += delegateResult
   sender.get.stop
   activeActors -= 1
   if(initializationReady && activeActors<=0) {
     clientActor ! sum
}  }
```

`initializationReady`, `activeActors` and `sum` are an instance variables rather than local variables, because these are used in two distinct parts of the partial function that is handed to `receive`. A test determines whether the calculation has completed and the result should be sent to the `clientActor`.
The SubScript version is about half the size:

```
live
  = ...
    << context: Context
    => var sum: Double = 0
       ( for(task <- context.tasks)
       & {!val actor=actorOf[Delegate].start
         actor ! DoTask(task) !}
         << d:Double
         => {sum += d; sender.get.stop}
         >>
       )
       {clientActor ! sum}
    >>
```

This version does not need to keep track of the state as recorded by `initializationReady` and `activeActors`. `sum` may now be a local variable, because everything happens in the `live` script. The delegate actors are created and started in a parallel loop; in the same loop the results are collected. When all branches of the loop have finished, the total computation is ready; then the result is sent to the `clientActor` as the final action.

[7] Based on https://github.com/yannart/ParallelPolynomialIntegral

7. RELATED WORK

Shivers[Shivers(1996)] argued that task specific sublanguages should be embedded in a syntactically extensible universal language. He extended the language Scheme this way with Unix like support for processes, including pipelines.
Linda [Carriero and Gelernter(1989)] is a coordination and communication model for tuple data stored in an associative memory. Processes store and retrieve these tuples; for the retrieval tuple fields may or may not be required to match specific values. The asynchronous tuple communication implies loosely coupled processes. In SubScript process communication is synchronous, but it is possible to spawn processes; when a send action is spawned this becomes in effect a partner for asynchronous communication. Moreover, the same kind of matching by field values is supported as in Linda; output values for retrieved fields are in both Linda and SubScript marked with question marks.
Futures [Flanagan and Felleisen(1999)] are constructs that act as proxies for values that are initially unknown, and of which the computation is thereafter ongoing or complete. These are available in libraries for languages such as Java and Scala. E.g. in an addition of two futures `x+y` the operands `x` and `y` may be computed in background threads; when the value of the result expression is needed, the results of the background threads are awaited. Thus simple code fragments express at the same time both concurrency and a computational relation. In SubScript these two are separated and comparable specifications are therefore less concise. On the other hand futures require an explicit call back style; the the control flow is in a data element (the Future) and its associated call back code. In SubScript the control flow is more explicit. The more complicated the computation dependencies are, the more useful the futures approach seems over SubScript.
The Orc programming language[Kitchin et al.(2009)Kitchin, Quark, Cook, and Misra] has a calculus with four types of combinators; two of these may be described as parallelism and failure continuation ("otherwise"); the other two are a kind of networking operators describing dataflow. Other idioms such as or-parallelism can be expressed in terms of combinators.
Grammar specification formalisms are closely related to ACP. They are applied in numerous parser generator languages, such as YACC [Johnson(1979)]. These interoperate to varying degrees with a base language; they yield parsers that are driven by tables rather than call graphs; therefore their speed is higher than SubScript. Also they offer more convenient means to express syntax and semantic parser actions. For SubScript it is still a big challenge to match the grammar specification power of the 43 year old YACC. On the other hand parser generator languages do not support parallelism and their usage is largely restricted to text parsing.
Scala Parser Combinators is a DSL in regular Scala that supports grammar specifications. A parser combinator specification may share some visual elements with SubScript parsers: carets and arrows, though these have a different meaning. Using plain Scala has both positive and negative sides: on the one hand standard Scala tools apply; there is no need to learn new syntax; on the other hand this makes some boilerplate code inevitable.
Toolbus[de Jong and Klint(2002)] is a coordination architecture based on ACP extended with data terms. Internal Toolbus processes may communicate data using standard ACP style. Toolbus focuses on control flow and data flow between Toolbus processes and external programs.

8. CONCLUSION

The main concepts for dataflow programming are already about half a century old, and still nearly absent in mainstream programming languages. This paper has shown how a nondeterministic and concurrent programming language may benefit from dataflow constructs. Ideas inspired by YACC appear to be well applicable to GUI controllers. Pipes, as known from Unix command shell language, may also ease software composition at the program level, as opposed to the operating system level. Actor programs may get a clearer control flow using nondeterministic and concurrent language constructs.
Performance may be an issue, in particular when the piping constructs are used between components for stream input, output and various kinds of filtering. Each communication action may involve thousands of machine instructions, so character by character communication will be slow. Possibly an optimizing SubScript virtual machine will be able to reduce the performance penalty.
An open source project[8] implements SubScript as a branch of the regular Scala compiler, bundled with a virtual machine and a library with scripts for Swing GUI events. The dataflow features that this paper highlights are at present being added to the implementation.

APPENDIX

A. REFERENCES

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

[Carl Hewitt(1973)] R. S. Carl Hewitt, Peter Bishop. Artificial intelligence a universal modular actor formalism for artificial intelligence, 1973.

[Carriero and Gelernter(1989)] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32:444–458, April 1989.

[de Jong and Klint(2002)] H. de Jong and P. Klint. Toolbus: The next generation. volume 2852 of *Lecture Notes in Computer Science*, pages 220–241. Springer, 2002.

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

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

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

[Johnson(1979)] S. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, 1979.

[Kitchin et al.(2009)Kitchin, Quark, Cook, and Misra] D. Kitchin, A. Quark, W. Cook, and J. Misra. The orc programming language, 2009.

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

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

[Odersky et al.(2008)Odersky, Spoon, and Venners] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, Mountain View, CA, 2008.

[Shivers(1996)] O. Shivers. A universal scripting framework or lambda: the ultimate "little language". In *Concurrency and Parallelism: Programming, Networking and Security*, pages 254–265. Springer-Verlag, 1996.

[van Delft()] A. van Delft. Subscript: Extending scala with the algebra of communicating processes. Scala Days 2012.

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