Scala-Gopher: CSP-style programming techniques with idiomatic Scala.

Ruslan Shevchenko
ruslan@shevchenko.kiev.ua
Kiev, Ukraine
note: scala-2016 OSS Talk

October 31, 2016

1 Introduction

Scala-gopher is a library-level implementation of process algebra [Communication Sequential Processes, see [2] as ususally enriched by π-calculus [5] naming primitives] in scala. In addition to support of a 'limbo/go-like' [8] [10] channels/goroutine programming style scala-gopher provides a set of operations following typical idiomatic scala.

First, let's review the fundamentals of a CSP model. The primary entities of this model are channels, coroutines and selectors. Coroutines are lightweight threads of execution, that can communicate with each other by passing messages between channels. Channels can be viewed as blocked multiproducer/multiconsumer queues. Sending message to unbuffered channel suspends producing coroutines until the moment when this message will have been read by some consumer. Buffered channels transfer control flow between sinks not on each message, but when internal channel buffer is full. In such way, communication via channel implicitly provides flow control functionality. At last, a selector statement is a way of coordination of several communication activities: like Unix select(2) system call, select statement suspends current coroutines until one of the actions (reading/writing to one of the channels in selector) will be possible.

Let's look at the simple example:

```
def nPrimes(n:Int):Future[List[Int]]=
{
   val in = makeChannel[Int]()
   val out = makeChannel[Int]()
   go {
     for(i <- 1 to Int.MaxValue) in.write(i)
   }
   go {
     select.fold(in){ (ch,s) =>
       s match {
         case p:ch.read => out.write(p)
                           ch.filter(_ % p != 0)
       }
     }
   }
   go {
     for(i <- 1 to n) yield out.read
   }
}
```

Here two channels and three goroutines are created. The first coroutine just generates consecutive numbers and sends one to channel `in`, second - accepts this sequence as the initial state and for each number that has been read from state channel, writes one to `out` and produces next step by filtering previous. The result of the fold is the `in` channel whith applied filters for each prime. The third coroutine just maps range to values to receive a list of first `n` primes in Future.

If we look at the sequence of steps during code evaluation, we will see at first generation of number, then checks in filters and then if a given number was prime - final output. Note, that goroutine is different from JVM thread of execution: sequential code chunks are executed in configurable executor service; switching between chunks does not use blocking operations.

2 Implementation of base constructs

2.1 Go: Translations of hight-order functions to asynchronous form.

The main entity of CSP is a 'process' which can be viewed as a block of code that handles specific events. In Go CSP processes are represented as goroutines (aka coroutines).

`go[X](x:X):Future[X]` is a thin wrapper arround SIP-22 async/await that does some preprocessing before async transformation:

- do transformation of hight-order function in async form.

  Let $f(A \Rightarrow B) \Rightarrow C$ is a hight-order function, that accepts other function $g : A \Rightarrow B$ as parameter. Let's say that g in f is ıinvocation-only if f does not store g in memory outside of f scope and doesn't return g as part of return value. The only two things that f can do with g is an invocation or passing it as a parameter to other invocation-only function. If we look at Scala collection API, we will see, that near all hight-order functions there are invocation-only.

  Now, if we have g which is invocation-only in f, and if we have a function $g' : (A \Rightarrow Future[B])$ let's build function $f' : (A \Rightarrow Future[B]) \rightarrow Future[C]$ that if $await(g') == await(g)$ then $await(f'(g')) == f(g))$ in the following way

  - f' translated to $await$(transformed-body(f))
  - $g(x)$ inside f translated to $await(g'(x))$
  - $h(g)$ translated to $await(h'(g'))$ if g is invocation-only in h.

  Scala-gopher contains asynchronous variants of predefined functions from Scala collection API, so it is possible to use asynchronous expressions inside a loop. For example, next code:

```
go {
    for(i <- 1 to n) yield out.read
}
```

  is transformed to

```
async{ await {
  (1 to n).mapAsync(i => async{ await{ out.aread } } )
} }
```

that after simplification step becomes

```
(1 to n).mapAsync(i => out.aread)
```

Using this approach allows overcoming the inconvenience of async/await by allowing programmers to use hight-order functions API inside asynchronous expression. Also, it is theoretically possible to generate asynchronous variants of API methods by transforming TASTY[4] representation of AST of synchronous versions. A similar technique is implemented in Nim [7] programming language, where we can generate both synchronous and asynchronous variants of a function from one definition.

- do transformation of defer statement. This is just an implementation of error handling mechanism.

2.2 Channels: callbacks organized as waits

Channels in CSP are two-sided pipes between processes; Channels messages not only pass information between goroutines but also coordinate process execution. In scala-gopher appropiative entities (`Input[A]` for reading and `Output[A]` for writing) implemented in fully asynchnious manner with help of callback-based interfaces:

```
trait Input[A]
{

   def  cbread[B](f:
            ContRead[A,B]=>Option[
                    ContRead.In[A]=>Future[Continuated[B]]
            ],
            ft: FlowTermination[B]): Unit
   ....
}
```

Here we can read argument type as protocol where each arrow is a step: f is called on opportunity to read and $ContRead[A,B] \Rightarrow Option[ContRead.In[A] \Rightarrow Future[B]]$ means that when reading is possible, we can ignore this opportunity (i.e. return None) or return handler that will consume value (or `end-of-input` or few other special cases) and return future to the next computation state.

Traditional synchronous API (i.e. method $read :\Rightarrow A$) can be used inside `go` and `async` statements; from 'normal' code we can use asynchronous variant: $aread :\Rightarrow Future[A]$.

Output interface is similar:

```
trait Output[A]
{

  def  cbwrite[B](f: ContWrite[A,B] => Option[
                   (A,Future[Continuated[B]])
                 ],
                 ft: FlowTermination[B]): Unit

}
```

Here f is called on opportunity to write and when we decide to use this opportunity, we must provide actual value to write and next step in the same way as with `Input`.

Inputs and outputs are composable as can be expected in a functional language and equipped by the usual set of stream combinators: filter, map, zip, fold, etc.

Channel is a combination of input and output. In addition to well-known buffered and unbuffered kinds of channels, scala-gopher provides some extra set of channels with different behavior and performance characteristics, such as the channel with growing buffer (a-la actor mailbox) for connecting loosely coupled processes or one-time channel based on `Promise`, which is automatically closed after sending one message.

2.3 Selectors: process composition as event generation

Mutually exclusive process composition (i.e. deterministic choice: $(a \rightarrow P) \Box (b \rightarrow Q)$ in original Hoar notation) usually represented in CSP-based languages as `select` statement, which looks like ordinary switch. In a typical

Go program exists often repeated code pattern: select inside endless loop inside go statement.

```
go {
  for{
    select{
     case c1 -> x :
           ......... // P
     case c2 <- y :
           ........  // Q
    }
  }
}
```

Appropriative expression in CSP syntax: $*[(c_1?x \rightarrow P)\Box(c_2!y \rightarrow Q)]$

Scala-gopher provides `select` pseudo-object that provides a set of higher-order pseudo-functions over channels, which accept syntax of partial function over channel events:

```
go {
  select.forever {
    case x: c1.read => ....  //P
    case y: c2.write => ....  //Q
  }
}
```

or version that must not be wrapped by `go` stamenet:

```
  select.aforever {
    case x: c1.read => ....  //P
    case y: c2.write => ....  //Q
  }
```

Under the hood, each such pseudo-function is built around a flow (sequence of `Continuated[_]` which represents the step of computations optionally bound to channel event). In unsugared form selector

```
  val selector = SelectorForever()
  selector.onRead(ch)((x,ft,ec) => ... ) // P after go-transform
  selector.onWrite(ch,y)((y,ft,ec) => ... ) // Q after go-transform
  selector.run()
```

We can maintain state inside a flow in a clear functional manner using 'fold' family of select functions:

```
def fibonacci(c: Output[Long], quit: Input[Boolean]): Future[(Long,Long)] =
   select.afold((0L,1L)) { case ((x,y),s) =>
    s match {
      case x: c.write => (y, x+y)
      case q: quit.read =>
                 select.exit((x,y))
    }
   }
```

Here we see the special syntax for tuple state. Also note, that `afold` macro assumes that `s match` must be the first statement in the argument pseudo-function. `select.exit` is used for returning result from the flow.

Events that we can check in select match statement are reading and writing of channels and select timeouts. In future we will think about extending the set of notifications - i.e. adding channel closing and overflow notifications, which are needed in some rare scenarios.

2.4 Transputer: an entity that encapsulates processing node.

The idea is to have an actor-like object, that encapsulates processing node: i.e., reads input data from the set of input ports; writes a result to the set of output ports and maintaines a local mutable state inside.

Example:

```
class Zipper[T] extends SelectTransputer
{

   val inX: InPort[T]
   val inY: InPort[T]

   val out: OutPort[(T,T)]

   loop {
     case x: inX.read =>
             val y = inY.read
```

```
            out write (x,y)
     case y: inY.read =>
            val x = inX.read
            out.write((x,y))
   }


}
```

Having a set of such objects, we can build complex systems as combinations of simple ones:

- $a + b$ - parallel execution;
- $replicate[A](n)$ - transputer replication, where we start in parallel n instances of A. Policy for port replication can be configured - from sharing appropriative channel by each port to distributing or duplication of signals to ditinguish each instance.

  ```
  val r = gopherApi.replicate[SMTTransputer](10)
     ( r.dataInput.distribute( (_.hashCode % 10 ) ).
        .controlInput.duplicate().
         out.share()
     )
  ```

  - here in `r` we have ten instances of `SMTTransputer`. If we send a message to `dataInput` it will be directed to one of the instances (according to the value of message's `hashcode`). If we send a message to the `controlInput`, it will be delivered to each instance; output channel will be shared between all instances.

Transputers can participate in error handling scenarios in the same way as actors: for each transputer, we can define recovery policy and supervisor.

2.5 Programming Techniques based on dynamic channels

Let's outline some programming techniques, well known in a Go world but not easily expressible in current mainstream Scala streaming libraries.

- Channel expression as an element of runtime state. Following this pattern allows a developer to maintain dynamics potentially recursive dataflows.

  Example: Imagine situation, where we need to distribute some tasks across a set of relative slow consumers and we need to spawn additional consumers on peak usage and free resources during the calm.

```
select.fold(output){ (out, s) => s match {
  case x:input.read =>
    select.once {
      case x:out.write =>
      case select.timeout =>
            control.distributeBandwidth match {
               case Some(newOut) => newOut.write(x)
                                    (out | newOut)
               case None => control.report("Can't increase bandwidth")
                                    out
            }
    }
  case select.timeout =>
    out match {
      case OrOutput(frs,snd) => snd.close
                                frs
      case _                 => out
    }
} }
```

  Here we can request additional channels from control and construct merged channel in the state of the fold. On read timeout, we can deconstruct merged channel back and free unused resources.

- Channel-based API where client supply channel where to pass reply

  Let we want provide API that must on request return some value to the caller. Instead of providing a method that will return a result on the stack we can provide endpoint channel, that will accept method arguments and channel where to return a result.

  Next example illustrates this idea:

```
trait Broadcast[T]
{
   val listener: Output[Channel[T]]
   val messages: Input[T]
}

class BroadcastImpl[T] extends Broadcast[T]
{

   val listener: Channel[Channel[T]] = makeChannel[Channel[T]]
   val messages: Channel[T] = makeChannel[]

   // private part
   case class Message(next:Channel[Message],value:T)

   select.afold(makeChannel[Message]) { (bus, s) =>
      s match {
         case v: message.read =>
                   val newBus = makeChannel[Message]
                   current.write(Message(newBus,v))
                   newBus
         case ch: listener.read =>
                   select.afold(bus) { (current,s) =>
                     s match {
                       case msg:current.read =>
                               ch.awrite(msg.value)
                               current.awrite(msg)
                               msg.next
                     }
                   }
                   current
     }
  }

}
```

Broadcast provides API for creation of listeners and sending messages to all listeners.

To register listener channel for receiving notification client sends this channel to newListener

The internal state contains message bus represented by a channel which is replaced during each new input message. Each listener spawns the process that reads messages from the current message bus.

3 Connection with other models

There are many stream libraries for Scala with different sets of tradeoffs. At one side of spectrum, we have clear streaming models like akka-streams[3] with comp set of composable operations and clear high-level functionality but lack of flexibility, at the other side - very flexible but low-level models like actors.

Scala-gopher provides uniform API that allows build systems from different parts of spectrum: it is possible to build dataflow graph in a declarative manner and connect one with a dynamic part.

The reactive isolates model[6] is close to scala-gopher model with dynamically-grown channel buffer (except that reactive isolates support distributed case). Isolate here corresponds to Transputer, Channel to Output and Events to gopher Input. Channels in reactive isolates are more limited: only one isolate that owns the channel can write to it when in the scala-gopher concept of channel ownership is absent.

Communicating Scala Objects[9] is a direct implementation of CSP model in Scala that allows building expressions in internal Scala DSL, closed to original Hoar notation with some extensions, like extended rendezvous for mapping input streams. Processes in CSO are not lightweight: each process requires Java thread which limits the scalability of this library until lightweight threading will be implemented on JVM level.

Subscript[1] is a Scala extension which adds to language new constructions for building process algebra expressions. Although extending language can afford fine-grained interconnection of process-algebra and imperative language notation in far perspective, now it makes CPA constructs a second-class citizen because we have no direct representation of process and event types in Scala type system.

4 Conclusion and future directions

Scala-gopher is a relatively new library that has not yet reached 1.0 state, but we have the early experience reports from using the library for building some helper components in an industrial software project.

In general, feedback is positive: developers enjoy a relatively simple mental model and ability to freely use asynchronous operations inside higher-order functions. So, we can recommend to made conversion of invocation-only functions into async form to be available into the async library itself.

The area that needs more work is an error handling in `go` statements: now `go` returns `Future` which can hold a result of the evaluation or an exception. If we ignore statement result then we miss handling of exception; from another side, it unlikely handle errors there, because we must allow a developer to implement own error processing. The workaround is to use different method name for calling statement in the context with ignored return value, but it is easy to mix-up this two names. We think, that right solution can be built on language level: we can bind handling of ignored value to type by providing appropriative implicit conversion or use special syntax for functions which needs

Also, we plan to extend model by adding notifications about channel-close and channel-overflow on write side that are needed in some relatively rare usage scenarios.

Support of distributed communications can be the next logical step in future development. In our opinion, practical approach will differ from implementing location-transparency support for existing API. Rather we will think to enrich model with new channel types and mechanisms for work in explicitly distributed environment.

Finally, we can say that CSP model can be nicely integrated with existing Scala concurrency ecosystem and have a place in existing zoo of concurrency models. Using scala-gopher allows developers to use well establishment elegant techniques, such as dynamic recursive data flows and channel-based API-s.

5 Literature

References

[1] André van Delft. "Dataflow Constructs for a Language Extension Based on the Algebra of Communicating Processes". In: *Proceedings of the 4th Workshop on Scala*. SCALA '13. Montpellier, France: ACM, 2013, 12:1–12:10. ISBN: 978-1-4503-2064-1. DOI: 10.1145/2489837.2489849. URL: http://doi.acm.org/10.1145/2489837.2489849.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. 1985.

[3] lightbend. *Akka-streams*. 2015. URL: http://doc.akka.io/docs/akka/2.4/scala/stream/index.html.

[4] Eugene Burmako Martin Odersky Dmitry Petrashko. *TASTY Reference Manual*. 2016. URL: https://docs.google.com/document/d/1h3KUMxsSSjyze05VecJGQ5H2yh7fNADtIf3chD3_wr0/.

[5] Robin Milner, Joachim Parrow, and David Walker. "A Calculus of Mobile Processes, I". In: *Inf. Comput.* 100.1 (Sept. 1992), pp. 1–40. ISSN: 0890-5401. DOI: 10.1016/0890-5401(92)90008-4. URL: http://dx.doi.org/10.1016/0890-5401(92)90008-4.

[6] Aleksandar Prokopec and Martin Odersky. "Isolates, Channels, and Event Streams for Composable Distributed Programming". In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Onward! 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 171–182. ISBN: 978-1-4503-3688-8. DOI: 10.1145/2814228.2814245. URL: http://doi.acm.org/10.1145/2814228.2814245.

[7] Andreas Rumpf. *Nim Programming Language*. Nov. 2015. URL: http://nim-lang.org.

[8] Phillip Stanley-Marbell. *Inferno Programming with Limbo*. John Wiley & Sons, 2003. ISBN: 978-0-470-84352-9.

[9] Bernard Sufrin. "Communicating Scala Objects". In: *Volume 66: Communicating Process Architectures 2008*. Concurrent Systems Engineering Series. 2008, pp. 35 –54. DOI: 10.3233/978-1-58603-907-3-35.

[10] *The Go Programming Language*. 2010. URL: https://golang.org.