Journal of Universal Computer Science, vol. 10, no. 7 (2004), 910-925
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © J.UCS

Coroutines in Lua

Ana Lúcia de Moura
(Catholic University of Rio de Janeiro, Brazil
ana@inf.puc-rio.br)

Noemi Rodriguez
(Catholic University of Rio de Janeiro, Brazil
noemi@inf.puc-rio.br)

Roberto Ierusalimschy
(Catholic University of Rio de Janeiro, Brazil
roberto@inf.puc-rio.br)

Abstract: After a period of oblivion, a renewal of interest in coroutines is being observed. However, most current implementations of coroutine mechanisms are restricted, and motivated by particular uses. The convenience of providing true coroutines as a general control abstraction is disregarded. This paper presents and discusses the coroutine facilities provided by the language Lua, a full implementation of the concept of asymmetric coroutines. It also shows that this powerful construct supports easy and succint implementations of useful control behaviors.

Key Words: coroutines, control structures, generators, Lua language
Category: D.3.3

1 Introduction

The concept of a coroutine is one of the oldest proposals for a general control abstraction. It is attributed to Conway [Conway, 1963], who described coroutines as "subroutines who act as the master program", and implemented this construct to simplify the cooperation between the lexical and syntactical analysers in a COBOL compiler. Marlin's doctoral thesis [Marlin, 1980], widely acknowledged as a reference for this mechanism, resumes the fundamental characteristics of a coroutine as follows:

- "the values of data local to a coroutine persist between successive calls";
- "the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage".

The aptness of the concept of a coroutine to express several useful control behaviors was perceived and explored during some years in a number of contexts, such as concurrent programming, simulation, text processing, artificial intelligence, and various kinds of data structures manipulation [Marlin, 1980, Pauli and Soffa, 1980]. Nevertheless, the convenience of providing a programmer with this powerful control abstraction has been disregarded by designers of general-purpose languages, with rare exceptions such as Simula [Birtwistle et al., 1976], BCPL [Moody and Richards, 1980], Modula-2 [Wirth, 1985], and Icon [Griswold and Griswold, 1996].

The absence of coroutine facilities in mainstream languages can be partly attributed to the lacking of an uniform view of this concept, which was never precisely defined. Moreover, most descriptions of coroutines found in the literature, Marlin's thesis included, are still based on Simula, a truly complex implementation of coroutines that contributed to the common misconception that coroutines are an "awkward" construct, difficult to manage and understand.

After a period of oblivion, we can now observe a renewal of interest in some forms of coroutines, notably in two different groups. The first group corresponds to developers of multitasking applications, who investigate the advantages of cooperative task management as an alternative to multithreading environments [Adya et al., 2002, Behren et al., 2003]. In this scenario, the concurrent constructs that support cooperative multitasking are usually provided by libraries or system resources like Window's *fibers* [Richter, 1997]. It is worth noticing that, although the description of the concurrency mechanisms employed in those works is no more than a description of the coroutine abstraction, the term coroutine is not even mentioned.

Another current resurgence of coroutines is in the context of scripting languages, notably Lua, Python and Perl. Python [Schemenauer et al., 2001] has recently incorporated a restricted form of coroutines that permits the development of simple *iterators*, or *generators*, but are not powerful enough to implement interesting features that can be written with true coroutines, including user-level multitasking. A similar mechanism is being proposed for Perl [Conway, 2000]. A different approach was followed by the designers of Lua, who decided on a full implementation of coroutines.

The purpose of this work is to present and discuss the coroutine facilities provided by Lua. Section 2 gives a brief introduction to the language and describes its coroutine facilities, providing an operational semantics for this mechanism. Section 3 illustrates the expressive power of Lua asymmetric coroutines by showing some relevant examples of their use. Section 4 discusses coroutines in some other languages. Section 5 presents our conclusions.

2 Lua Coroutines

Lua [Ierusalimschy et al., 1996, Figueiredo et al., 1996] is a lightweight scripting language that supports general procedural programming with data description facilities. It is dynamically typed, lexically scoped, interpreted from bytecodes, and has automatic memory management with garbage collection. Lua was originally designed, and is typically used, as an *extension* language, embedded in a host program.

Lua was designed, from the beginning, to be easily integrated with software written in C, C++, and other conventional languages. Lua is implemented as a small library of C functions, written in ANSI C, and compiles virtually unmodified in all currently available plataforms. Along with the Lua interpreter, this library provides a set of functions (the C API) that enables the host program to communicate with the Lua environment. Through this API a host program can, for instance, read and write Lua variables and call Lua functions. Besides allowing Lua code to extend a host application, the API also permits the extension of Lua itself by providing facilities to register C functions to be called by Lua. In this sense, Lua can be regarded as a language framework for building domain-specific languages.

Lua implements the concept of *asymmetric* coroutines, which are commonly denoted as *semi-symmetric* or *semi-coroutines* [Marlin, 1980, Dahl et al., 1972]. Asymmetric coroutine facilities are so called because they involve two types of control transfer operations: one for (re)invoking a coroutine and one for suspending it, the latter returning control to the coroutine invoker. An asymmetric coroutine can be regarded as subordinate to its caller, the relationship between them being similar to that between a called and a calling routine. A different control discipline is implemented by *symmetric* coroutine facilities, which provide a single transfer operation for switching control to the indicated coroutine. Because symmetric coroutines are capable of passing control between themselves, they are said to operate at the same hierarchical level. The following arguments justify why Lua offers asymmetric coroutines, instead of providing symmetric facilities or both mechanisms.

It has been argued that symmetric and asymmetric coroutines have no equivalent power, and that general-purpose coroutine facilities should provide both constructs [Marlin, 1980, Pauli and Soffa, 1980]. However, it is easy to demonstrate that symmetric coroutines can be expressed by asymmetric facilities (see Appendix A). Therefore, no expressive power is lost if only asymmetric coroutines are provided. (Actually, implementing asymmetric coroutines on top of symmetric facilities is equally simple). Implementing both abstractions only complicates the semantics of the language. In Simula, for instance, the introduction of semicoroutines led to problems in understanding the details of coroutine sequencing, and several efforts to describe the semantics of Simula coroutines were shown to be inconsistent [Marlin, 1980].

Since expressive power is not an issue, preserving two of the main characteristics of the Lua, *simplicity* and *portability*, constitutes the main reason for implementing asymmetric facilities. Most programmers nowadays have already been exposed to the the concept of a *thread*, which, like a coroutine, represents a line of execution that can be interrupted and later resumed at the point it was suspended. Nevertheless, coroutine mechanisms are frequently described as difficult to understand. In fact, handling explicitly the sequencing between *symmetric* coroutines is not an easy task, and requires a considerable effort from the programmer. Even experienced programmers may have difficulties in understanding the control flow of a program that employs a moderate number of symmetric coroutines. On the other hand, asymmetric coroutines truly behave like *routines*, in the sense that control is always transfered back to their callers. Since even novice programmers are familiar with the concept of a routine, control sequencing with asymmetric coroutines seems much simpler to manage and understand, besides allowing the development of more structured programs. A similar argument is used in proposals of *partial continuations* that, like asymmetrical coroutines, can be composed like regular functions [Danvy and Filinski, 1990, Queinnec and Serpette, 1991, Hieb et al., 1994].

The other motivation for implementing asymmetric coroutines was the need to preserve Lua's ease of integration with its host language (C) and also its portability. Lua and C code can freely call each other; therefore, an application can create a chain of nested function calls wherein the languages are interleaved. Implementing a symmetric facility in this scenario imposes the preservation of C state when a Lua coroutine is suspended. This preservation is only possible if a coroutine facility is also provided for C; but a portable implementation of coroutines for C cannot be written. On the other hand, we do not need coroutine facilities in C to support Lua asymmetric coroutines; all that is necessary is a restriction that a coroutine cannot yield while there is a C function in that coroutine stack.

2.1 Lua Coroutine Facilities

Lua coroutine facilities provide three basic operations: `create`, `resume`, and `yield`. Like in most Lua libraries, these functions are packed in a global table (table `coroutine`).

Function `coroutine.create` creates a new coroutine, and allocates a separate stack for its execution. It receives as argument a function that represents the main body of the coroutine and returns a coroutine reference. Creating a coroutine does not start its execution; a new coroutine begins in suspended state with its *continuation point* set to the first statement in its main body. Quite often, the argument to `coroutine.create` is an anonymous function, like this:

```
co = coroutine.create(function() ... end)
```

Lua coroutines are first-class values; they can be stored in variables, passed as arguments and returned as results. There is no explicit operation for deleting a

Lua coroutine; like any other value in Lua, coroutines are discarded by garbage collection.

Function `coroutine.resume` (re)activates a coroutine, and receives as its required first argument the coroutine reference. Once resumed, a coroutine starts executing at its continuation point and runs until it suspends or terminates. In either case, control is transfered back to the coroutine's invocation point.

A coroutine suspends by calling `coroutine.yield`; in this case, the coroutine's execution state is saved and the corresponding call to `coroutine.resume` returns immediately. By implementing a coroutine as a separate stack, Lua allows calls to `coroutine.yield` to occur even from inside nested Lua functions (i.e., directly or indirectly called by the coroutine main function). The next time the coroutine is resumed, its execution will continue from the exact point where it suspended.

A coroutine terminates when its main function returns; in this case, the coroutine is said to be *dead* and cannot be further resumed. A coroutine also terminates if an error occurs during its execution. When a coroutine terminates normally, `coroutine.resume` returns `true` plus any values returned by the coroutine main function. In case of errors, `coroutine.resume` returns `false` plus an error message.

Like `coroutine.create`, the auxiliary function `coroutine.wrap` creates a new coroutine, but instead of returning the coroutine reference, it returns a function that resumes the coroutine. Any arguments passed to that function go as extra arguments to `resume`. The function also returns all the values returned by `resume`, except the status code. Unlike `coroutine.resume`, the function does not catch errors; any error that ocurrs inside a coroutine is propagated to its caller. A simple implementation of function `wrap` using the basic functions `coroutine.create` and `coroutine.resume` is illustrated next:

```
function wrap(f)
  local co = coroutine.create(f)
  return function(v)
           status, val = coroutine.resume(co, v)
           if status then return val
           else error(val)
           end
         end
end
```

Lua provides a very convenient facility to allow a coroutine and its caller to exchange data. As we will see later, this facility is very useful for the implementation of *generators*, a control abstraction that produces a sequence of values, each at a time. As an illustration of this feature, let us consider the coroutine created by the following code:

```
co = coroutine.wrap(function(a)
                      local c = coroutine.yield(a + 2)
                      return c * 2
                    end)
```

The first time a coroutine is activated, any extra arguments received by the correspondent invocation are passed to the coroutine main function. If, for instance, our sample coroutine is activated by calling

```
b = co(20)
```

the coroutine function will receive the value 20 in `a`. When a coroutine suspends, any arguments passed to function `yield` are returned to its caller. In our example, the coroutine result value 22 (`a + 2`) is received by the assignment `b = co(20)`.

When a coroutine is reactivated, any extra arguments are returned by the corresponding call to function `yield`. Proceeding with our example, if we reactivate the coroutine by calling

```
d = co(b + 1)
```

the coroutine local variable `c` will get the value 23 (`b + 1`).

Finally, when a coroutine terminates, any values returned by its main function go to its last invocation point. In this case, the result value 46 (`c * 2`) is received by the assignment `d = co(b + 1)`.

2.2 An Operational Semantics for Lua Asymmetric Coroutines

In order to clarify the details of Lua asymmetric coroutines, we now develop an operational semantics for this mechanism. This operational semantics is partially based on the semantics for *subcontinuations* provided in [Hieb et al., 1994]. We start with the same core language, a call-by-value variant of the λ-calculus extended with assignments. The set of expressions in this core language (e) is in fact a subset of Lua expressions: constants (c), variables (x), function definitions, function calls, and assignments:

$$e \;\rightarrow\; c \mid x \mid \lambda x.e \mid e\,e \mid x := e$$

Expressions that denote values (v) are constants and functions:

$$v \;\rightarrow\; c \mid \lambda x.e$$

A store θ, mapping variables to values, is included in the definition of the core language to allow side-effects:

$$\theta : variables \;\rightarrow\; values$$

The following *evaluation contexts* (C) and *rewrite rules* define a left-to-right, applicative order semantics for evaluating the core language.

$$C \;\rightarrow\; \Box \mid C\,e \mid v\,C \mid x := C$$

$$\langle C[x],\, \theta\rangle \Rightarrow \langle C[\theta(x)],\, \theta\rangle \tag{1}$$

$$\langle C[(\lambda x.e)v],\, \theta\rangle \Rightarrow \langle C[e],\, \theta[x \leftarrow v]\rangle,\; x \notin \mathrm{dom}(\theta) \tag{2}$$

$$\langle C[x := v],\, \theta\rangle \Rightarrow \langle C[v],\, \theta[x \leftarrow v]\rangle,\; x \in \mathrm{dom}(\theta) \tag{3}$$

Rule 1 states that the evaluation of a variable fills the context with its associated value in θ. Rule 2 describes the evaluation of applications; in this case, α-substitution is assumed in order to guarantee that a new variable x is inserted into the store. In rule 3, which describes the semantics of assignments, it is assumed that the variable already exists in the store (i.e., it was previously introduced by an abstraction).

In order to incorporate asymmetric coroutines into the language, we extend the set of expressions with *labels*, *labeled expressions* and coroutine operators:

$$e \;\rightarrow\; c \mid x \mid \lambda x.e \mid e\,e \mid x := e \mid l \mid l : e \mid \mathit{create}\,e \mid \mathit{resume}\,e\,e \mid \mathit{yield}\,e$$

Because labels are used to reference coroutines, we include them in the set of expressions that denote values

$$v \;\rightarrow\; c \mid \lambda x.e \mid l$$

and extend the definition of the store, allowing mappings from labels to values:

$$\theta : (\mathit{variables} \cup \mathit{labels}) \;\rightarrow\; \mathit{values}$$

Finally, the definition of evaluation contexts must incorporate the new expressions:

$$\begin{aligned} C \;\rightarrow\; & \Box \mid C\,e \mid v\,C \mid x := C \mid \\ & \mathit{create}\,C \mid \mathit{resume}\,C\,e \mid \mathit{resume}\,l\,C \mid \mathit{yield}\,C \mid l : C \end{aligned}$$

We can now develop rewrite rules that describe the semantics of Lua coroutines. Two types of evaluation contexts are used: *full* contexts (C) and *subcontexts* (C'). A subcontext is an evaluation context that does not contain labeled contexts ($l : C$). It corresponds to an innermost active coroutine (i.e., a coroutine wherein no nested coroutine occurs).

$$\langle C[\mathit{create}\,v],\, \theta\rangle \Rightarrow \langle C[l],\, \theta[l \leftarrow v]\rangle,\; l \notin \mathrm{dom}(\theta) \tag{4}$$

$$\langle C[\mathit{resume}\,l\,v],\, \theta\rangle \Rightarrow \langle C[l : \theta(l)\,v],\, \theta[l \leftarrow \perp]\rangle \tag{5}$$

$$\langle C_1[l : C_2'[\mathit{yield}\,v]],\, \theta\rangle \Rightarrow \langle C_1[v],\, \theta[l \leftarrow \lambda x.C_2'[x]]\rangle \tag{6}$$

$$\langle C[l : v],\, \theta\rangle \Rightarrow \langle C[v],\, \theta\rangle \tag{7}$$

Rule 4 describes the action of creating a coroutine. It creates a new label to represent the coroutine and extends the store with a mapping from this label to the coroutine main function.

Rule 5 shows that the `resume` operation produces a labeled expression, which corresponds to a coroutine continuation obtained from the store. This continuation is invoked with the extra argument passed to `resume`. In order to prevent the coroutine to be reactivated, its label is mapped to an invalid value, represented by $\perp$.

Rule 6 describes the action of suspending a coroutine. The evaluation of the *yield* expression must occur within a labeled subcontext (C_2'), resulting from the evaluation of the *resume* expression that invoked the coroutine; this guarantees that a coroutine returns control to its correspondent invocation point. The continuation of the suspended coroutine is saved in the store. This continuations is represented by a function whose main body is created from the corresponding subcontext. The argument passed to `yield` becomes the result value obtained by resuming the coroutine.

The last rule defines the semantics of coroutine termination, and shows that the value returned by the coroutine main body becomes the result value obtained by the last activation of the coroutine.

3 Programming With Lua Asymmetric Coroutines

Lua asymmetric coroutines are an expressive construct that permits the implementation of several control paradigms. By implementing this abstraction, Lua is capable of providing convenient features for a wide range of applications, while preserving its distinguishing economy of concepts. This section describes the use of Lua asymmetrical coroutines to implement two useful features: *generators* and *cooperative multitasking.*

3.1 Lua Coroutines as Generators

A *generator* is a control abstraction that produces a sequence of values, returning a new value to its caller for each invocation. A typical use of generators is to implement *iterators*, a related control abstraction that allows traversing a data structure independently of its internal implementation [Liskov et al., 1977]. Besides the capability of keeping state, the possibility of exchanging data when transfering control makes Lua coroutines a very convenient facility for implementing iterators.

To illustrate this kind of use, the following code implements a classical example: an iterator that traverses a binary tree in pre-order. Tree nodes are represented by Lua tables containing three fields: `key`, `left` and `right`. Field key stores the node value (an integer); fields left and right contain references to the node's respective children.

```
function preorder(node)
  if node then
    preorder(node.left)
    coroutine.yield(node.key)
    preorder(node.right)
  end
end

function preorder_iterator(tree)
  return coroutine.wrap(function() preorder(tree) end)
end
```

Function preorder_iterator receives as argument a binary tree's root node and returns an iterator that successively produces the values stored in the tree nodes. The possibility of yielding from inside nested calls allows an elegant and concise implementation of the tree iterator. The traversal of the tree is performed by an auxiliary recursive function (preorder) that yields the produced value directly to the iterator's caller. The end of a traversal is signalled by producing a nil value, implicitly returned by the iterator's main function when it terminates.

An example of use of the binary tree iterator, the merge of two trees, is shown next. Function merge receives two binary trees as arguments. It begins by creating iterators for the two trees (it1 and it2) and collecting their smallest elements (v1 and v2). The while loop prints the smallest value, and reinvokes the correspondent iterator for obtaining its next element, continuing until the elements in both trees are exhausted.

```
function merge(t1, t2)
  local it1 = preorder_iterator(t1)
  local it2 = preorder_iterator(t2)
  local v1 = it1()
  local v2 = it2()

  while v1 or v2 do
    if v1 ~= nil and (v2 == nil or v1 < v2) then
      print(v1); v1 = it1()
    else
      print(v2); v2 = it2()
    end
  end
end
```

Generators are also a convenient construct for implementing goal-oriented programming, as in solving Prolog queries [Clocksin and Mellish, 1981] and doing pattern-matching problems. In this scenario, a problem or *goal* is either a *primitive* goal or a *disjunction* of alternative solutions, or subgoals. These subgoals are, in turn, *conjunctions* of goals that must be satisfied in succession, each of them contributing a partial outcome to the final result. In pattern-matching problems, for instance, string literals are primitive goals, alternative patterns are disjunctions of subgoals and sequences of patterns are conjunctions of goals. The unification process in Prolog is an example of a primitive goal, a Prolog relation is a disjunction and Prolog rules are conjunctions of goals. Solving a goal then typically involves implementing a backtracking mechanism that successively tries each alternative solution until an adequate result is found.

Lua asymmetric coroutines used as generators simplifies the implementation of this type of control behavior, avoiding the complex bookeeping code required to manage explicit backtrack points. Wrapping a goal in a Lua coroutine allows a backtracker, implemented as a simple loop construct, to successively retry (*resume*) a goal it until an adequate result is found. A primitive goal can be defined as a function that *yields* at each invocation one of its successful results. A disjunction can be implemented by a function that sequentially invokes its alternative goals. A conjunction of two subgoals can be defined as a function that iterates on the first subgoal, invoking the second one for each produced outcome. It is worth noticing that, again, the possibility of yielding from inside nested calls is essential for this concise, straightforward implementation.

3.2 User-Level Multitasking

The aptness of coroutines as a concurrent construct was perceived by Wirth, who introduced them in Modula-2 [Wirth, 1985] as a basic facility to support the development of concurrent programs. Due mainly to the introduction of the concept of *threads*, and its adoption in modern mainstream languages, this suitable use of coroutines is, unfortunately, currently disregarded.

A language with coroutines does not require additional concurrency constructs to provide multitasking facilities: just like a thread, a coroutine represents a unit of execution that has its private data and control stack, while sharing global data and other resources with other coroutines. However, while the concept of a thread is typically associated with *preemptive* multitasking, coroutines provide an alternative concurrency model which is essentially *cooperative*. A coroutine must explicitly request to be suspended to allow another coroutine to run.

The development of correct multithreading applications is widely acknowledged as a complex task. In some contexts, like operating systems and real-time applications, where timely responses are essential, preemptive task scheduling is unavoidable; in this case, programmers with considerable expertise are responsible for implementing adequate synchronization strategies. The timing requirements of most concurrent applications, though, are not critical. Moreover, application developers have, usually, little or no experience in concurrent programming. In this scenario, ease of development is a relevant issue, and a cooperative multitasking environment, which eliminates conflicts due to race conditions and minimizes the need for synchronization, seems much more appropriate.

Implementing a multitasking application with Lua coroutines is straightforward. Concurrent tasks can be modeled by Lua coroutines. When a new task is created, it is inserted in a list of live tasks. A simple task dispatcher can be implemented by a loop that continuously iterates on this list, resuming the live tasks and removing the ones that have finished their work (this condition can be signalled by a predefined value returned by the coroutine main function to the dispatcher). Occasional fairness problems, which are easy to identify, can be solved by adding suspension requests in time-consuming tasks.

The only drawback of cooperative multitasking arises when using blocking operations; if, for instance, a coroutine calls an I/O operation and blocks, the entire program blocks until the operation completes, and no other coroutine has a chance to proceed. This situation is easily avoided by providing auxiliary functions that initiate an I/O request and yield, instead of blocking, when the operation cannot be immediately completed. A complete example of a concurrent application implemented with Lua coroutines, including non-blocking facilities, can be found in [Ierusalimschy, 2003].

4 Coroutines in Programming Languages

The best-known programming language that incorporates a coroutine facility is Simula [Birtwistle et al., 1976, Dahl et al., 1972], which also introduced the concept of semi-coroutines. In Simula, coroutines are organized in an hierarchy that is dynamically set up. The relationship between coroutines at the same hierarchical level is symmetric; they exchange control between themselves by means of `resume` operations. When a Simula coroutine is activated by means of a `call` operation, it becomes hierachically subordinated to its activator, to which it can transfer control back by calling `detach`. Because Simula coroutines can behave either as symmetric or semi-symmetric coroutines (and, sometimes, as both), their semantics is extremely complicated, and even experienced Simula programmers may have difficulties in understanding the control flow in a program that makes use of both constructs.

BCPL, a systems programming language widely used in the 1970's and the oldest ancestor of C, is another example of a language that incorporates coroutine facilities [Moody and Richards, 1980]. Like Simula, the BCPL coroutine mechanism provides both asymmetric (`Callco`/`Cowait`) and symmetric (`Resumeco`) control transfer operations. The designers of the BCPL coroutine mechanism remarked, though, that a scheme involving only asymmetric facilities would have a wide range of applications.

Modula-2 [Wirth, 1985] incorporates symmetric coroutines as a basic construct for implementing concurrent processes. However, the potential of coroutine constructs to implement other forms of control behaviors is not well explored in this language.

The iterator abstraction was originally proposed and implemented by the designers of CLU [Liskov et al., 1977]. Because a CLU iterator preserves state between succesive calls, they described it as a coroutine. However, CLU iterators are not first-class objects, and are limited to a `for` loop construct that can invoke exactly one iterator. Parallel traversals of two or more collections are not possible. Sather iterators [Murer et al., 1996], inspired by CLU iterators, are also confined to a single call point within a loop construct. The number of iterators invoked per loop is not restricted as in CLU, but if any iterator terminates, the loop is terminated. Although traversing multiple collections in a single loop is possible with Sather iterators, aynchronous traversals, as required for merging two binary trees, have no simple solution.

In Python [Schemenauer et al., 2001] a function that contains an `yield` statement is called a *generator function*. When called, a generator function returns a first-class object that can be resumed at any point in a program. However, a Python generator can be suspended only when its control stack is at the same level that it was at creation time; in other words, only the main body of a generator can yield. A similar facility has been proposed for Perl 6 [Conway, 2000]: the addition of a new return command, also called `yield`, which preserves the execution state of the subroutine in which it's called.

Python generators and similar constructs complicate the structure of recursive or more sophisticated generators. If items are produced within nested calls or auxiliary functions, it is necessary to create an hierarchy of auxiliary generators that "yield" in succession until the generator's original call point is reached. Moreover, this type of construct is far less expressive than true coroutines; for instance, it does not support the implementation of user-level multithreading.

Stackless Python [Laird, 2000] is an alternative implementation of Python that was initially motivated by the provision of continuations as a basic control construct [Tismer, 2000]. The focus of this implementation, though, has been recently redirected to the direct support of generators, coroutines and microthreads. The direct implementation of all these constructs clearly indicates that the language designer does not recognize the power of coroutines as a single unifying concept.

Icon's goal-directed evaluation of expressions [Griswold and Griswold, 1996] is a powerful language paradigm where backtracking is supported by another restricted form of coroutines, named *generators* — expressions that may produce multiple values. Besides providing a collection of built-in generators, Icon also supports user-defined generators — user-defined procedures that `suspend` instead of returning. Although not limited to an specific construct, Icon generators are confined to the expression in which they are contained, and are invoked only by explicit iteration and goal-directed evaluation. Icon generators per se, then, are not powerful enough to provide for programmer-defined control structures. To support this facility, Icon provides *co-expressions*, first-class objects that wrap an expression and an environment for its evaluation, so that the expression can be explicitly resumed at any place. Co-expressions are, actually, an implementation of asymmetric coroutines.

5 Conclusions

In this article we have described the concept of asymmetric coroutines as implemented by the language Lua. We have also demonstrated the generality of this abstraction by showing that a language that provides true asymmetrical coroutines has no need to implement additional constructs to support several useful control behaviors.

It is not difficult to show that the expressive power of asymmetric coroutines is equivalent to that of one-shot *subcontinuations* [Hieb et al., 1994] and other forms of *partial continuations* [Queinnec, 1993] that, differently from traditional continuations, are non-abortive and can be composed like regular functions. [Danvy and Filinski, 1990], [Queinnec and Serpette, 1991] and [Sitaram, 1993] demonstrated that partial continuations provide more concise and understandable implementations of the classical applications of traditional continuations, such as generators, backtracking and multitasking. We have shown in this paper that these same applications can be equally easily expressed with asymmetrical coroutines. This is not a coincidence; actually, partial continuations and asymmetrical coroutines have several similarities, which we are exploring in a development of our work.

Despite its expressive power, the concept of a continuation is difficult to manage and understand, specially in the context of procedural programming. Asymmetrical coroutines have equivalent power, are arguably easier to implement and fits nicely in procedural languages.

Acknowledgements

This work was partially supported by grants from CNPq–Brazil. The authors would also like to thank the anonymous referees for their helpful comments.

References

[Adya et al., 2002] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Doucer, J. R. (2002). Cooperative Task Management without Manual Stack Management. In *Proceedings of USENIX 2002 Annual Technical Conference*, Monterey, California.

[Behren et al., 2003] Behren, R., Condit, J., and Brewer, E. (2003). Why Events are a Bad Idea (for high-concurrency servers). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii.

[Birtwistle et al., 1976] Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1976). *Simula Begin*. Studentlitteratur.

[Clocksin and Mellish, 1981] Clocksin, W. and Mellish, C. (1981). *Programming in Prolog*. Springer-Verlag.

[Conway, 2000] Conway, D. (2000). RFC 31: Subroutines: Co-routines. `http://dev.perl.org/perl6/rfc/31.html`.

[Conway, 1963] Conway, M. (1963). Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7).

[Dahl et al., 1972] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). Hierarchical program structures. In *Structured Programming*. Academic Press, Second edition.

[Danvy and Filinski, 1990] Danvy, O. and Filinski, A. (1990). Abstracting control. In *LFP'90 ACM Symposium on Lisp and Functional Programming*.

[Figueiredo et al., 1996] Figueiredo, L. H., Ierusalimshcy, R., and Celes, W. (1996). Lua: an extensible embedded language. *Dr Dobb's Journal*, 21(12).

[Griswold and Griswold, 1996] Griswold, R. and Griswold, M. (1996). *The Icon Programming Language*. Peer-to-Peer Communications, ISBN 1-57398-001-3, Third edition.

[Hieb et al., 1994] Hieb, R., Dybvig, R., and Anderson III, C. W. (1994). Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110.

[Ierusalimschy, 2003] Ierusalimschy, R. (2003). *Programming in Lua*. Lua.org, ISBN 85-903798-1-7.

[Ierusalimschy et al., 1996] Ierusalimschy, R., Figueiredo, L. H., and Celes, W. (1996). Lua-an extensible extension language. *Software: Practice & Experience*, 26(6).

[Laird, 2000] Laird, C. (2000). Introduction to Stackless Python. `http://www.onlamp.com/pub/a/python/2000/10/4/stackless-intro.html`.

[Liskov et al., 1977] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. (1977). Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8).

[Marlin, 1980] Marlin, C. D. (1980). *Coroutines: A Programming Methodology, a Language Design and an Implementation*. LNCS 95, Springer-Verlag.

[Moody and Richards, 1980] Moody, K. and Richards, M. (1980). A coroutine mechanism for BCPL. *Software: Practice & Experience*, 10(10).

[Murer et al., 1996] Murer, S., Omohundro, S., Stoutamire, D., and Szyperski, C. (1996). Iteration abstraction in Sather. *ACM Transactions on Progamming Languages and Systems*, 18(1).

[Pauli and Soffa, 1980] Pauli, W. and Soffa, M. L. (1980). Coroutine behaviour and implementation. *Software: Practice & Experience*, 10.

[Queinnec, 1993] Queinnec, C. (1993). A library of high level control operators. *ACM SIGPLAN Lisp Pointers*, VI(4).

[Queinnec and Serpette, 1991] Queinnec, C. and Serpette, B. (1991). A dynamic extent control operator for partial continuations. In *POPL'91 Eighteenth Annual ACM Symposium on Principles of Programming Languages*.

[Richter, 1997] Richter, J. (1997). *Advanced Windows*. Microsoft Press, Third edition.

[Schemenauer et al., 2001] Schemenauer, N., Peters, T., and Hetland, M. (2001). PEP 255 Simple Generators. `http://www.python.org/peps/pep-0255.html`.

[Sitaram, 1993] Sitaram, D. (1993). Handling control. In *ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation*.

[Tismer, 2000] Tismer, C. (2000). Continuations and Stackless Python. In *Proceedings of the 8th International Python Conference*, Arlington, VA.

[Wirth, 1985] Wirth, N. (1985). *Programming in Modula-2.* Springer-Verlag, Third, corrected edition.

A Implementing Symmetric Coroutines

The following code provides a Lua extension library that supports the creation of symmetric coroutines and their control transfer discipline:

```
coro = {}
coro.main = function() end
coro.current = coro.main

-- creates a new coroutine
function coro.create(f)
  return coroutine.wrap(function(val)
                           return nil, f(val)
                         end)
end

-- transfers control to a coroutine
function coro.transfer(k, val)
  if coro.current ~= coro.main then
    return coroutine.yield(k, val)
  else
    -- dispatching loop
    while k do
      coro.current = k
      if k == coro.main then
        return val
      end
      k, val = k(val)
    end
    error("coroutine ended without transfering control...")
  end
end
```

The basic idea in this implementation is to simulate symmetric transfers of control between Lua coroutines with pairs of yield/resume operations and an auxiliary "dispatching loop". In order to allow coroutines to return control to the main program, table `coro` (which packs the symmetric coroutine facility) provides a field (`main`) to represent the main program.

When a coroutine, or the main program, wishes to transfer control, it calls `coro.transfer`, passing the coroutine to be (re)activated; an extra argument defined for this operation allows coroutines to exchange data. If the main program is currently active, the dispatching loop is executed; if not, function `transfer` uses `coroutine.yield` to reactivate the dispatcher, which acts as an intermediary in the switch of control (and data) between the coroutines. When control is to be transfered to the main program, function `transfer` returns.