Revisiting Coroutines

Ana Lúcia de Moura and Roberto Ierusalimschy
Computer Science Department – PUC-Rio
Rua M. S. Vicente 225, 22453-900
Rio de Janeiro RJ, Brazil
{*ana, roberto*}*@inf.puc-rio.br*

PUC-RioInf.MCC15/04 June, 2004

Abstract: This paper defends the revival of coroutines as a general control abstraction. After proposing a new classification of coroutines, we introduce the concept of full asymmetric coroutines and provide a precise definition for it through an operational semantics. We then demonstrate that full coroutines have an expressive power equivalent to one-shot continuations and one-shot partial continuations. We also show that full asymmetric coroutines and one-shot partial continuations have many similarities, and therefore present comparable benefits. Nevertheless, coroutines are easier implemented and understood, specially in the realm of procedural languages. Finally, we provide a collection of programming examples that illustrate the use of full asymmetric coroutines to support direct and concise implementations of several useful control behaviors, including cooperative multitasking.

Keywords: control structures, coroutines, continuations

Resumo: Este artigo defende o resgate de corotinas como uma poderosa abstração de controle. Nele propomos uma nova classificação para corotinas e introduzimos o conceito de corotinas assimétricas completas, formalizado através de uma semântica operacional. Demonstramos então que corotinas completas tem poder expressivo equivalente ao de continuações *one-shot* e continuações parciais *one-shot*. Mostramos também que corotinas assimétricas completas e continuações parciais *one-shot* têm diversas semelhanças e, consequentemente, apresentam benefícios similares. Corotinas, porém, são mais facilmente implementadas e compreendidas, especialmente num contexto de linguagens procedurais. Finalmente, apresentamos uma coleção de exemplos de programação que ilustram o uso de corotinas assimétricas completas para implementar, de forma sucinta e elegante, diversas estruturas de controle interessantes, incluindo concorrência cooperativa.

Palavras-chave: estruturas de controle, corotinas, continuações

1 Introduction

The concept of coroutines was introduced in the early 1960s and constitutes one of the oldest proposals of a general control abstraction. It is attributed to Conway, who described coroutines as "subroutines who act as the master program", and implemented this construct to simplify the cooperation between the lexical and syntactical analyzers in a COBOL compiler [Conway 1963]. The aptness of coroutines to express several useful control behaviors was widely explored during the next twenty years in several different contexts, including simulation, artificial intelligence, concurrent programming, text processing, and various kinds of data-structure manipulation [Knuth 1968; Marlin 1980; Pauli and Soffa 1980]. Nevertheless, designers of general-purpose languages have disregarded the convenience of providing a programmer with this powerful control construct, with rare exceptions such as Simula [Birtwistle et al. 1980], BCPL [Moody and Richards 1980], Modula-2 [Wirth 1985], and Icon [Griswold and Griswold 1983].

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. 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".

This description of coroutines corresponds to the common perception of the concept, but leaves open relevant issues with respect to a coroutine construct. Apart from the capability of keeping state, we can identify three main issues that distinguish coroutine facilities:

- the control-transfer mechanism, which can provide *symmetric* or *asymmetric* coroutines;
- whether coroutines are provided in the language as *first-class* objects, which can be freely manipulated by the programmer, or as constrained constructs;
- whether a coroutine is a *stackful* construct, i.e., whether it is able to suspend its execution from within nested calls.

Depending on the intended use for the coroutine mechanism, particular solutions for the preceding issues were adopted. As a consequence, quite different implementations of coroutines were developed, such as Simula's

and Modula's coroutine facilities, Icon's generators and co-expressions, and, more recently, Python generators [Schemenauer et al. 2001]. Although all these constructs satisfy Marlin's general characterization of coroutines, they provide significantly different degrees of expressiveness[1].

Besides the absence of a precise definition, the introduction of first-class continuations also greatly contributed to the virtual end of research interest in coroutines as a general control abstraction. Unlike coroutines, first-class continuations have a well-defined semantics and are widely acknowledged as an expressive construct that can be used to implement several interesting features, including generators, exception handling, backtracking [Felleisen 1985; Haynes 1987], multitasking at the source level [Dybvig and Hieb 1989; Wand 1980], and also coroutines [Haynes et al. 1986]. However, with the exception of Scheme [Kelsey et al. 1998], some implementations of ML [Harper et al. 1991], and an alternative implementation of Python [Tismer 2000], first-class continuations are not usually provided in programming languages.

A relevant obstacle to the incorporation of continuations in a language is the difficulty to provide an efficient implementation of this construct. This difficulty is mainly due to the need of supporting multiple invocations of a continuation, which usually involves copying a captured continuation before it is modified [Hieb et al. 1990]. The observation that in most contexts continuations are actually invoked only once motivated Bruggeman et al. [1996] to introduce *one-shot* continuations, which are limited to a single invocation and thus eliminate the copying overhead associated with multi-shot continuations. One-shot continuations can replace multi-shot continuations in practically all their useful applications. Specially in the implementation of multitasking, one-shot continuations can provide significant performance benefits when compared to multi-shot continuations [Bruggeman et al. 1996].

Apart from efficiency issues, the concept of a continuation as a representation of the rest of a computation is difficult to manage and understand, specially in the context of procedural languages. The abortive nature of a continuation invocation complicates considerably the structure of programs; even experienced programmers may have difficulties to understand the control flow of continuation-intensive applications. The convenience of limiting the extent of continuations and localizing the effects of their control operators motivated the introduction of *partial continuations* [Felleisen 1988; Johnson and Duggan 1988] and the proposal of a series of constructs based on this concept [Queinnec 1993]. Unlike traditional continuations, partial continuations represent only a "continuation slice" [Queinnec 1993], or the continuation of a subcomputation [Hieb et al. 1994]. Partial continuations are not abortive; they are composed with the current continuation and thus behave like regular functions. Danvy and Filinski [1990], Queinnec and Serpette [1991], and Sitaram [1993] demonstrated that control constructs based

[1]In this paper we use the concept of expressiveness as defined by Felleisen [1990].

on partial continuations can provide more concise and understandable implementations of the classical applications of continuations, such as generators, backtracking, and multitasking. In all these applications, the restriction imposed to one-shot continuations can be applied to partial continuations, allowing more efficient implementations of the constructs. Despite their advantages over traditional continuations, partial continuations are not provided in common implementations of programming languages, even in the realm of Scheme.

Another significant reason for the absence of coroutines in modern languages is the current adoption of *multithreading* as a *de facto* standard for concurrent programming. In the last years several research efforts have been dedicated to alternative concurrency models that can support more efficient and less error-prone applications, such as event-driven programming and cooperative multitasking. Nevertheless, mainstream languages like Java and C# still provide threads as their primary concurrency construct.

The purpose of this paper is to defend the revival of coroutines as a powerful control abstraction, which fits nicely in procedural languages and can be easily implemented and understood. We argue and demonstrate that, contrary to common belief, coroutines are not *far* less expressive than continuations. Instead, when provided as first-class objects and implemented as stackful constructs — that is, when a *full* coroutine mechanism is implemented — coroutines have equivalent power to that of one-shot continuations. Based on similar arguments as presented in the proposals of partial continuations mechanisms — easiness to manage and understand, and support for more structured applications — we specifically defend full *asymmetric* coroutines as a convenient construct for language extensibility.

The remainder of this paper is organized as follows. Section 2 proposes a classification of coroutine mechanisms based on the three issues mentioned earlier, and discusses their influence on the usefulness of a coroutine facility. Section 3 provides a formal description of our concept of full asymmetric coroutines and illustrates it with an example of a general-purpose programming language that implements this mechanism. In section 4 we show that full asymmetric coroutines can provide not only symmetric coroutine facilities but also one-shot continuations and one-shot partial continuations. Section 5 contains a collection of programming examples that use full asymmetric coroutines to provide direct implementations of several useful control behaviors, including multitasking. Finally, section 6 summarizes the paper and presents our conclusions.

2 A Classification of Coroutines

The capability of keeping state between successive calls constitutes the general and commonly adopted description of a coroutine construct. However,

we observe that the various implementations of coroutine mechanisms differ widely with respect to their convenience and expressive power. In this section we identify and discuss the three issues that most notably distinguish coroutine mechanisms and influence their usefulness.

2.1 Control Transfer Mechanism

A well-known classification of coroutines concerns the control-transfer operations that are provided and distinguishes the concepts of *symmetric* and *asymmetric* coroutines. Symmetric coroutine facilities provide a single control-transfer operation that allows coroutines to explicitly pass control between themselves. Asymmetric coroutine mechanisms (more commonly denoted as *semi-symmetric* or *semi* coroutines [Dahl et al. 1972]) provide two control-transfer operations: one for invoking a coroutine and one for suspending it, the latter returning control to the coroutine invoker. While symmetric coroutines operate at the same hierarchical level, an asymmetric coroutine can be regarded as subordinate to its caller, the relationship between them being somewhat similar to that between a called and a calling routine.

Coroutine mechanisms to support concurrent programming usually provide symmetric coroutines to represent independent units of execution, like in Modula-2. On the other hand, coroutine mechanisms intended for implementing constructs that produce sequences of values typically provide asymmetric coroutines. Examples of this type of construct are *iterators* [Liskov et al. 1977; Murer et al. 1996] and *generators* [Griswold and Griswold 1983; Schemenauer et al. 2001]. The general-purpose coroutine mechanisms implemented by Simula and BCPL provide both types of control transfer. In the absence of a formal definition of coroutines, Simula's mechanism, a truly complex implementation of coroutines, was practically adopted as a reference for a general-purpose coroutine mechanism and greatly contributed to the common misconception that symmetric and asymmetric coroutines have no equivalent power. However, it is easy to demonstrate that we can express any of these constructs in terms of the other; therefore, a general-purpose coroutine mechanism can provide either symmetric or asymmetric coroutines. Providing both constructs only complicates the semantics of the mechanism, with no increase in its expressive power.

Although equivalent in terms of expressiveness, symmetric and asymmetric coroutines are not equivalent with respect to ease of use. Handling and understanding the control flow of a program that employs even a moderate number of symmetric coroutines may require a considerable effort from a programmer. On the other hand, asymmetric coroutines behave like routines, in the sense that control is always transfered back to their invokers. Since even novice programmers are familiar with the concept of a routine, control sequencing is simpler to manage and understand. Moreover, asym-

metric coroutines allow the development of more structured programs.

2.2 First-class versus constrained coroutines

An issue that considerably influences the expressive power of a coroutine mechanism is whether coroutines are provided as first-class objects. In some implementations of coroutines, typically intended for particular uses, coroutine objects are constrained within a textual bound and cannot be directly manipulated by the programmer. An example of this restricted form of coroutine is the *iterator* abstraction, which was originally proposed and implemented by the designers of CLU to permit the traversal of data structures independently of their internal representation [Liskov et al. 1977]. Because a CLU iterator preserves state between successive calls, they described it as a coroutine; actually, an iterator fits Marlin's general characterization of coroutines. However, CLU iterators are confined within a `for` loop that can invoke exactly one iterator. This restriction imposes a considerable limitation to the use of the construct; parallel traversals of two or more data collections, for instance, 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 terminates. Although traversing multiple collections in a single loop is possible, asynchronous traversals, as required for merging data collections, have no simple solution. Icon's goal-directed evaluation of expressions [Griswold and Griswold 1983] is an interesting language paradigm where backtracking is supported by another constrained 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, implemented by procedures that suspend instead of returning. Despite not being limited to a specific construct, Icon generators are confined within an expression and can only be invoked by explicit iteration or goal-directed evaluation. Icon generators are easier to use than CLU and Sather iterators, but they are not powerful enough to provide for programmer-defined control structures. This facility is only provided when coroutines are implemented as first-class objects, which can be freely manipulated by the programmer and invoked at any place. First-class coroutines are provided, for instance, by Icon *co-expressions* and the coroutine facilities implemented by Simula, BCPL, and Modula-2.

2.3 Stackfulness

Stackful coroutine mechanisms allow coroutines to suspend their execution from within nested functions; the next time the coroutine is resumed, its execution continues from the exact point where it suspended. Stackful coroutine mechanisms are provided, for instance, by Simula, BCPL, Modula-2, Icon,

and also by CLU and Sather's iterators.

A currently observed resurgence of coroutines is in the context of scripting languages, notably Python and Perl. In Python [Schemenauer et al. 2001], a function that contains an `yield` statement is called a *generator function*. When called, this function returns an object that can be resumed at any point in a program, so it behaves as an asymmetric coroutine. Despite constituting a first-class object, a Python generator is not a stackful construct; it can only suspend its execution 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 suspend. A similar facility has been proposed for Perl 6 [Conway 2000]: the addition of a new type of return command, also called `yield`, which preserves the execution state of the subroutine in which it is called.

Python generators and similar non-stackful constructs permit the development of simple iterators or generators but complicate the structure of more elaborate implementations. As an example, if items are produced within recursive or auxiliary functions, it is necessary to create an hierarchy of auxiliary generators that yield in succession until the original invocation point is reached. This type of generator is also not powerful enough to implement user-level multitasking.

2.4 Full coroutines

Based on the preceding discussion, we can argue that, according to our classification, two issues determine the expressive power of a coroutine facility: whether coroutines are first-class objects and whether they are stackful constructs. In the absence of these facilities, a coroutine mechanism cannot support several useful control behaviors, notably multitasking, and, therefore, does not provide a general control abstraction. We then introduce the concept of a *full* coroutine as a first-class, stackful object, which, as we will demonstrate later, can provide the same expressiveness as obtained with one-shot continuations.

Full coroutines can be either symmetric or asymmetric; the selection of a particular control-transfer mechanism does not influence their expressive power. However, asymmetric coroutines are more easily managed and can support more succinct implementations of user-defined constructs. Therefore, we believe that full asymmetric coroutines mechanisms provide a more convenient control abstraction than symmetric facilities.

3 Full Asymmetric Coroutines

The purpose of this section is to provide a precise definition for our concept of full asymmetric coroutines. We begin by introducing the basic operators

of this model of coroutines. We then formalize the semantics of these operators by developing an operational semantics for a simple language that incorporates them. Finally, we provide an example of a programming language that implements a full asymmetric coroutine mechanism that closely follows our proposed semantics. We will use this language in the programming examples later presented in this paper.

3.1 Coroutine Operators

Our model of full asymmetric coroutines has three basic operators: *create*, *resume*, and *yield*. The operator *create* creates a new coroutine. It receives a procedural argument, which corresponds to the coroutine main body, and returns a reference to the created coroutine. 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.

The operator *resume* (re)activates a coroutine. It receives as its first argument a coroutine reference, returned from a previous *create* operation. Once resumed, a coroutine starts executing at its saved continuation point and runs until it suspends or its main function terminates. In either case, control is transfered back to the coroutine's invocation point. When its main function terminates, the coroutine is said to be *dead* and cannot be further resumed.

The operator *yield* suspends a coroutine execution. The coroutine's continuation point is saved so that the next time the coroutine is resumed, its execution will continue from the exact point where it suspended.

Our coroutine operators provide a convenient facility to allow a coroutine and its invoker to exchange data. The first time a coroutine is activated, a second argument given to the operator *resume* is passed as an argument to the coroutine main function. In subsequent reactivations of a coroutine, that second argument becomes the result value of the operator *yield*. On the other hand, when a coroutine suspends, the argument passed to the operator *yield* becomes the result value of the operator *resume* that activated the coroutine. When a coroutine terminates, the value returned by its main function becomes the result value of its last reactivation.

3.2 Operational Semantics

In order to formalize our concept of full asymmetric coroutines, we now develop an operational semantics for this mechanism. The many similarities between asymmetric coroutines and *subcontinuations* (which we will discuss in Section 4.3) allow us to base this semantics on the operational semantics of subcontinuations described by Hieb et al. [1994]. We start with the same core language, a call-by-value variant of the λ-calculus extended with assignments. In this core language, the set of expressions (denoted by e)

includes 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:

$$\theta : \mathit{variables} \;\rightarrow\; \mathit{values}$$

The evaluation of the core language is defined by a set of rewrite rules that are applied to expression–store pairs until a value is obtained. *Evaluation contexts* [Felleisen and Friedman 1986] are used to determine, at each step, the next subexpression to be evaluated. The evaluation contexts (C) defined for the core language are

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

The preceding definition specifies a left-to-right evaluation of applications, because the argument can only be in an evaluation context when the term in the function position is a value. The rewrite rules for evaluating the core language are given next:

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

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

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

Rule 1 states that the evaluation of a variable results in its stored value in θ. Rule 2 describes the evaluation of applications; in this case, α-substitution is assumed in order to guarantee that a fresh 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* (l), *labeled expressions* ($l : e$), and the coroutine operators:

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

In our extended language, we use labels as references to coroutines, and labeled expressions to represent a currently active coroutine. As we will see later, labeling a coroutine context allows us to identify the coroutine being suspended when the operator *yield* is evaluated. Because labels are used to

reference coroutines, we must include them in the set of expressions that denote values:

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

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

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

The definition of evaluation contexts must include the new expressions. In this new definition we have specified a left-to-right evaluation for the operator *resume*, since only when its first argument (a coroutine reference) has been reduced to a label value its extra argument is examined:

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

We actually use two types of evaluation contexts: *full* contexts (denoted by C) and *subcontexts* (denoted by 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).

The rewrite rules that describe the semantics of the coroutine operators are given next:

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

$$\langle C[resume\,l\,v],\, \theta\rangle \;\Rightarrow\; \langle C[l : \theta(l)\,v],\, \theta[l \leftarrow \bot]\rangle \quad (5)$$

$$\langle C_1[l : C_2'[yield\,v]],\, \theta\rangle \;\Rightarrow\; \langle C_1[v],\, \theta[l \leftarrow \lambda x.C_2'[x]]\rangle \quad (6)$$

$$\langle C[l : v],\, \theta\rangle \;\Rightarrow\; \langle C[v],\, \theta\rangle \quad (7)$$

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 $\bot$.

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

The last rule defines the semantics of coroutine termination, and shows that the value returned by the coroutine main function becomes the result value obtained by the last activation of the coroutine. The mapping of the coroutine label to $\perp$, established when the coroutine was resumed, prevents the reactivation of a dead coroutine.

3.3 An example of a full asymmetric coroutine facility

Lua [Ierusalimschy et al. 1996; Ierusalimschy 2003] is a scripting language widely used in the game industry, an application domain where cooperative multitasking is a typical control behavior. Since its version 5.0 Lua provides a coroutine facility that closely follows the semantics we have just described.

3.3.1 An Overview of Lua

Lua is a light-weight language that supports general procedural programming with data description facilities. It is dynamically typed, lexically scoped, and has automatic memory management.

Lua supports eight basic value types: *nil*, *boolean*, *number*, *string*, *userdata*, *thread*, *function*, and *table*. The types *nil*, *boolean*, *number*, and *string* have usual meanings. The type *userdata* allows arbitrary C data to be stored in Lua variables. The type *thread* represents an independent thread of control and is used to implement coroutines.

Functions in Lua are first-class values: they can be stored in variables, passed as arguments to other functions, and returned as results. Lua functions are always anonymous; the syntax

```
function foo(x) ... end
```

is merely a syntactical sugar for

```
foo = function (x) ... end
```

Tables in Lua are associative arrays and can be indexed with any value; they may be used to represent ordinary arrays, symbol tables, sets, records, etc. In order to support a convenient representation of records, Lua uses a field name as an index and provides `a.name` as syntactic sugar for `a["name"]`. Lua tables are created by means of *constructor expressions*. The simplest constructor (`{}`), creates a new, empty table. Table constructors can also specify initial values for selected fields as in {`x = 1, y = 2`}.

Variables in Lua can be either *global* or *local*. Global variables are not declared and are implicitly given an initial `nil` value. Local variables are lexically scoped and must be explicitly declared.

Lua provides an almost conventional set of statements, similar to those in Pascal or C, including assignments, function calls, and traditional control structures (`if`, `while`, `repeat` and `for`). Lua also supports some not so conventional features such as multiple assignments and multiple results.

3.3.2 Lua Coroutines

Lua coroutine facilities implement our concept of full asymmetric coroutines [Moura et al. 2004]. Following the semantics of this mechanism, three basic operations are provided: `create`, `resume`, and `yield`, Like in most Lua libraries, these functions are packed in a global table (table `coroutine`).

Function `coroutine.create` allocates a separate Lua stack for the new coroutine. It receives as argument a Lua function that represents the main body of the coroutine and returns a coroutine reference (a value of type *thread*). Quite often, the argument to `coroutine.create` is an anonymous function, like this:

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

Like functions, 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.

Functions `coroutine.resume` and `coroutine.yield` closely follow the semantics of the operators *resume* and *yield* described before. However, because Lua functions can return multiple results, this facility is also provided by Lua coroutines. This means that function `coroutine.resume` can receive a variable number of extra arguments, which are all returned by the corresponding call to function `coroutine.yield`. Likewise, when a coroutine suspends, the corresponding call to function `coroutine.resume` returns all the arguments passed to `coroutine.yield`.

Like `coroutine.create`, the auxiliary function `coroutine.wrap` creates a new coroutine, but instead of returning the coroutine itself, it returns a function that, when called, resumes the coroutine. Any arguments passed to that function go as extra arguments to `resume`. The function also returns all the values returned by `coroutine.resume`, except the first one (a boolean error code). Usually, function `coroutine.wrap` is simpler to use than `coroutine.create`; it provides exactly what is typically needed: a function to resume a coroutine. Therefore, in all the programming examples provided in this paper we will be using `coroutine.wrap`.

As an illustration of Lua coroutines, let us consider a classical example: an iterator that traverses a binary tree in pre-order, shown in Figure 1. In this example, 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_iterator` receives as argument a binary tree's root node and returns an iterator that successively produces the values stored in the tree. The possibility of yielding from inside nested calls allows a concise implementation of the tree iterator: the traversal of the tree is performed by an auxiliary recursive function (`preorder`) that yields the produced value

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

-- create an iterator
function preorder_iterator(tree)
  return coroutine.wrap(function()
                          preorder(tree)
                          return nil
                        end)
end
```

Figure 1: A binary tree iterator implemented with Lua coroutines

directly to the iterator's caller. The end of a traversal is signalled by a `nil` value, returned by the iterator's main function when it terminates.

Figure 2 shows an example of use of the binary tree iterator: merging two binary trees. Function `merge` receives as arguments the two trees' root nodes. It begins by creating iterators for the 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
```

Figure 2: Merging two binary trees

4 Expressing Alternative Control Structures

It is a common belief that coroutines are far less expressive than first-class continuations (e.g., [Friedman et al. 1984]) and, also, that asymmetric coroutines are less powerful than symmetric coroutines. In this section we contradict these misconceptions by showing that a language that incorporates full asymmetric coroutines can easily provide not only symmetric coroutines but also one-shot continuations and one-shot partial continuations. Therefore, any sort of control structure implemented by these constructs can be provided by full asymmetric coroutines.

4.1 Symmetric Coroutines

The basic characteristic of symmetric coroutine facilities is the provision of a single control-transfer operation that allows coroutines to pass control explicitly among themselves. Marlin [1980] and Pauli and Soffa [1980] argued that symmetric and asymmetric coroutines have no equivalent power and that general coroutine facilities should provide both constructs. However, it is easy to demonstrate that we can provide any of these mechanisms using the other; therefore, no expressive power is lost if only asymmetric coroutines are provided in a language.

The implementation of symmetric coroutines on top of asymmetric facilities is straightforward. Symmetrical transfers of control between asymmetric coroutines can be easily simulated with pairs of yield–resume operations and an auxiliary dispatching loop that acts as an intermediary in the switch of control between the two coroutines. When a coroutine wishes to transfer control, it *yields* to the dispatching loop, which in turn *resumes* the coroutine that must be reactivated.

The code shown in Figure 3 illustrates this mechanism by providing a Lua library (`coro`) that supports the creation of symmetric coroutines and their control-transfer discipline. In order to allow coroutines to also transfer control to the main program, table `coro` provides a field (`main`) to represent it, simulating a coroutine reference. Another auxiliary field (`current`) is used to store a reference to the currently active coroutine.

When a coroutine, or the main program, wishes to transfer control, it calls function `coro.transfer`, passing the coroutine to be (re)activated; an extra argument provided to this function allows coroutines to exchange data. If the main program is currently active, the dispatching loop is executed; if not, function `coroutine.yield` is called to reactivate the dispatcher. When control is to be transfered to the main program, function `coro.transfer` returns.

In our implementation, we followed Modula's semantics of coroutine termination, which specifies that the termination of a coroutine without an explicit transfer of control constitutes a run-time error [Wirth 1985]. In or-

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

-- function to create a new coroutine
function coro.create(f)
  local co = function(val)
               f(val)
               error("coroutine ended")
             end
  return coroutine.wrap(co)
end

-- function to transfer control to a coroutine
function coro.transfer(co, val)
  if coro.current == coro.main then
    return coroutine.yield(co, val)
  end

  -- dispatching loop
  while true do
    coro.current = co
    if co == coro.main then
      return val
    end
    co, val = co(val)
  end
end
```

Figure 3: Implementing symmetric coroutines facilities

der to implement this semantics, function `coro.create` wraps the coroutine body in a function that issues an error to terminate the main program when the coroutine terminates.

4.2 One-shot Continuations

A *continuation* represents the rest of a computation from a given point in the computation. When they are provided as first-class objects, as in Scheme [Kelsey et al. 1998] and some implementations of ML [Harper et al. 1991], continuations can be used to implement a wide variety of control structures and thus represent a powerful tool for language extensibility.

In Scheme, the procedure `call/cc` causes the current continuation to be packaged as a first-class object. This captured continuation is then passed to the argument of `call/cc`, which must be a procedure of one argument. If this procedure returns without invoking the continuation, its returned

value becomes the value of the application of `call/cc`. If, at any time, the captured continuation is invoked with a value, this value is immediately returned to the continuation of the original `call/cc` application.

Although conventional first-class continuation mechanisms allow a continuation to be invoked multiple times, in virtually all their useful applications continuations are invoked only once. Motivated by this fact, Bruggeman et al. [1996] introduced the concept of *one-shot* continuations and the control operator `call/1cc`. One-shot continuations differ from multi-shot continuations in that it is an error to invoke a one-shot continuation more than once, either implicitly (by returning from the procedure passed to `call/1cc`) or explicitly (by invoking the continuation created by `call/1cc`).

The implementation of one-shot continuations described by Bruggeman et al. [1996] reveals the many similarities between this mechanism and symmetric coroutines. In this implementation, the control stack is represented as a linked list of *stack segments*, which are structured as stacks of *frames* or *activation records*. When a one-shot continuation is captured, the current stack segment is "encapsulated" in the continuation and a fresh stack segment is allocated to replace the current stack segment. In terms of symmetric coroutines, this corresponds to creating a new coroutine and transferring control to it. When a one-shot continuation is invoked, the current stack segment is discarded and control is returned to the saved stack segment. This is exactly what happens if the new coroutine, at any time, transfers control back to its creator.

The similarities between one-shot continuations and symmetric coroutines allow us to provide a concise implementation of `call/1cc` using the symmetric coroutine facility described in Section 4.1. This implementation is shown in Figure 4.

4.3 One-shot Subcontinuations

Despite their expressive power, traditional continuations, either multi-shot or one-shot, are difficult to use; except for some trivial examples, they complicate considerably the structure of programs [Felleisen 1988; Danvy and Filinski 1990; Queinnec and Serpette 1991]. Most of the complexity involved in the use of continuations arise from the fact that they represent the *whole* rest of a computation. The need to constrain the extent of continuations and localize the effects of control operators motivated the proposal of several control abstractions based on the concept of partial continuations [Queinnec 1993]. The essence of these abstractions is that the invocation of a captured partial continuation does not abort the current continuation; instead, partial continuations can be *composed* like regular functions.

Subcontinuations [Hieb et al. 1994] are an example of a partial continuation mechanism. A subcontinuation represents the rest of an independent partial computation (a *subcomputation*) from a given point in that subcom-

```
function call1cc(f)
  -- save the continuation "creator"
  local ccoro = coro.current

  -- invoking the continuation transfers control
  -- back to its creator
  local cont = function(val)
    if ccoro == nil then
      error("one shot continuation called twice")
    end
    coro.transfer(ccoro, val)
  end

  -- when a continuation is captured,
  -- a new coroutine is created and dispatched
  local val
  val = coro.transfer(coro.create(function()
                                    local v = f(cont)
                                    cont(v)
                                  end))

  -- when control is transfered back, the continuation
  -- was "shot" and must be invalidated
  ccoro = nil

  -- the value passed to the continuation
  -- is the return value of call1/cc
  return val
end
```

Figure 4: Implementing one-shot continuations with symmetric coroutines

putation. The operator `spawn` establishes the base, or root, of a subcomputation. It takes as argument a procedure (the subcomputation) to which it passes a *controller*. If the controller is not invoked, the result value of `spawn` is the value returned by the procedure. If the controller is invoked, it captures and aborts the continuation from the point of invocation back to, and including, the root of the subcomputation. The procedure passed to the controller is then applied to that captured subcontinuation. A controller is only valid when the corresponding root is in the continuation of the program. Therefore, once a controller has been applied, it will only be valid again if the subcontinuation is invoked, reinstating the subcomputation.

Like one-shot continuations and symmetric coroutines, one-shot subcontinuations and full asymmetric coroutines have many similarities. Full asymmetric coroutines can be regarded as independent subcomputations. Invoking a subcomputation controller is similar to suspending an asymmetric

coroutine. Invoking a one-shot subcontinuation corresponds to resuming an asymmetric coroutine. Like subcontinuations, asymmetric coroutines can be composed: they behave like regular functions, always returning control to their invoker. The main difference between subcontinuations and full asymmetric coroutines, and also between subcontinuations and other types of partial continuations, is that the reified subcontinuation is not restricted to the innermost subcomputation. Instead, a subcontinuation extends from the controller invocation point up to the root of the invoked controller, and may include several nested subcomputations.

The similarities between one-shot subcontinuations and full asymmetric coroutines allow us to express one-shot subcontinuations in terms of Lua coroutines and provide the implementation of the operator `spawn` shown in Figure 5. When function `spawn` is invoked, a Lua coroutine (`subc`) is created to represent the subcomputation. This coroutine's main body invokes `spawn`'s functional argument (`f`), passing to it a locally defined function that implements the subcomputation controller. Variable `valid_controller` indicates if it is legal to invoke the controller; its value is `true` only when the correspondent coroutine is active. The controller function creates a subcontinuation by suspending the coroutine; variable `shot` indicates whether this subcontinuation was invoked (i.e., whether the coroutine was resumed). When the coroutine is resumed, function `controller` returns to its caller, reactivating the subcomputation from the controller invocation point. The argument passed to the subcontinuation (returned by `coroutine.yield`) becomes the result value of the controller invocation.

Function `subK` is responsible for spawning and reinstating the subcomputation; it does so by resuming the correspondent coroutine. If the subcomputation ends without invoking the controller, the coroutine main body returns to its invocation point the result value obtained from calling `f` and a `nil` value (see line 6 in Figure 5). In this case, function `subK` terminates (line 28) and `spawn` returns to its caller the value returned by `f`. When the controller is invoked, the coroutine invocation (line 24) gets the argument passed to the controller (a function to be applied to the subcontinuation) and also a reference to the invoked controller. The controller reference, passed as the second argument to `coroutine.yield` (line 13), allows us to simulate a subcontinuation composed by an arbitrary number of nested subcomputations[2]. When a coroutine suspends, the returned controller reference is checked to verify if it belongs to the current scope (line 32). If it does, the function passed to the controller is applied to the subcontinuation. If not, it means that the invoked controller corresponds to an outer subcomputation, so the current coroutine calls `coroutine.yield` to reactivate it (line

[2]This behavior is also provided by variants of some partial-continuation mechanisms that use *marks* [Queinnec and Serpette 1991] or *tags* [Sitaram 1993] to specify the context up to which a partial continuation is to be reified.

```
 1 function spawn(f)
 2   local controller, valid_controller, shot, subK
 3
 4   -- a coroutine represents a subcomputation
 5   local subc = coroutine.wrap(function()
 6                                 return f(controller), nil
 7                               end)
 8
 9   -- this function implements the subcomputation controller
10   function controller(fc)
11     if not valid_controller then error("invalid controller") end
12     shot = false
13     val = coroutine.yield(fc, controller)
14     shot = true
15     return val
16   end
17
18   -- this function spawns/reinstates a subcomputation
19   function subK(v)
20     if shot then error("subcontinuation called twice") end
21
22     -- invoke a subcontinuation
23     valid_controller = true
24     local ret, c = subc(v)
25     valid_controller = false
26
27     -- subcomputation terminated ?
28     if c == nil then return ret
29
30     -- if the local controller was invoked, apply
31     --  the function to the subcontinuation
32     elseif c == controller then return ret(subK)
33
34     -- the invoked controller corresponds to
35     -- an outer subcomputation
36     else
37       val = coroutine.yield(ret, c)
38       return subK(val)
39     end
40   end
41
42   -- spawn the subcomputation
43   shot = false
44   return subK()
45 end
```

Figure 5: Implementing `spawn`

37). This process is repeated until the controller root is reached, allowing us to include all the suspended subcomputations in the captured subcontinuation. Symmetrically, invoking this subcontinuation will successively resume the suspended coroutines until the original controller invocation point is reached.

4.4 Efficiency issues

Haynes et al. [1986] demonstrated that continuations can be used to implement coroutines. Sitaram [1994] showed that coroutines can also be expressed in terms of partial continuations. We have just shown that full asymmetric coroutines can be used to express both one-shot continuations and one-shot partial continuations, which allow us to argue that full asymmetric coroutines have equivalent expressive power to those abstractions. However, expressing one-shot continuations and subcontinuations with coroutines and the reverse operations may not be equivalent in terms of efficiency.

In a simple implementation of one-shot continuations as described by Bruggeman et al. [1996], the creation of a continuation involves the conversion of the current stack segment into a continuation object and the allocation a new stack segment to replace it. When a one-shot continuation is invoked, the current segment is discarded and control is returned to the saved stack segment. The creation of a coroutine also involves the allocation of a separate stack. The actions of suspending and resuming a coroutine are just a little more expensive than regular function calls.

In our implementation of one-shot continuations, the creation of a single coroutine — i.e., a single stack "segment" — was sufficient to implement a continuation. Therefore, with a language that implements full coroutines we can provide one-shot continuation mechanisms that perform as efficiently as a simple direct implementation of this abstraction. On the other hand, the implementation of coroutines with continuations, as developed by Haynes et al. [1986], typically requires the capture of a new continuation each time a coroutine is suspended. This implementation thus involves the allocation of a new stack segment for each control transfer and, hence, performs much less efficiently and uses more memory than a direct implementation of coroutines.

Hieb et al. [1994] described a possible implementation of subcontinuations that uses a stack of labeled stacks. To ensure efficiency, this stack is represented by a stack of label-address pairs, with each address pointing to a stack segment stored elsewhere. Invoking `spawn` results in the addition of a new empty stack to the stack of labeled stacks; to this new stack a label is assigned in order to associate it with its correspondent controller. When the controller is invoked, all the stacks down to the one with the associated label are removed from the stack of labeled stacks, and packaged into a subcontinuation. When the subcontinuation is invoked, its saved stacks are pushed onto the current stack of labeled stacks.

In our implementation of one-shot subcontinuations, the cost of spawning a subcomputation (i.e., the creation and activation of a coroutine) is equivalent to that of the proposed implementation of `spawn`. When a subcontinuation involves a single subcomputation (the more usual case), the capture and invocation of a subcontinuation can perform at least as efficiently as the described direct implementation of subcontinuations. In the more complicated case, where a subcontinuation includes several nested subcomputations, the successive resumptions of the involved subcomputations impose some overhead, but with a cost no much higher than a succession of function calls. On the other hand, the implementation of coroutines with subcontinuations also requires the capture of a subcontinuation for each control transfer and, so, is arguably less efficient than a simple direct implementation of coroutines.

Kumar et al. [1998] described an implementation of one-shot subcontinuations in terms of threads. Their basic idea is somewhat similar to ours: a subcomputation is represented by a child thread, which is created when `spawn` is invoked. The parent thread is then put to sleep, waiting on a *done* condition, which is associated to the subcomputation controller. When the controller is invoked, the child thread wakes up the root thread by signalling the correspondent *done* condition, and then suspends its execution by creating and waiting on a *continue* condition. When a subcontinuation is invoked, the correspondent thread is woken by signalling its *continue* condition, and the invoker is put to sleep waiting on the controller's *done* condition. Besides the use of conditions to allow the suspension and resumption of threads (which, differently from coroutines, cannot explicitly transfer control), an additional synchronization mechanism (implemented with a *mutex*) is required to prevent a spawned child thread to signal the *done* condition before the parent thread is put to sleep[3].

The implementation of subcontinuations with threads does not involve the successive suspensions and resumptions of nested subcomputations that the use of coroutines requires. However, the use of threads introduces a considerable complexity and overhead, due to the need of synchronization mechanisms. Moreover, the capture of a subcontinuation requires the creation of a new condition, and the allocation of its associated structures. Besides being more efficient in the the more general case, implementing subcontinuations with coroutines is simpler and arguably less error-prone.

[3]This is actually a simplified description of the implementation shown in [Kumar et al. 1998]. We have considered only the requirements for an implementation of non-concurrent subcontinuations.

5 Programming With Full Asymmetric Coroutines

In the previous section we showed that a language with full asymmetric coroutines can easily provide one-shot continuations and one-shot partial continuations and, therefore, all control behaviors that can be implemented with those abstractions. In this section we complement our demonstration of the expressiveness of full asymmetric coroutines by providing succinct and elegant implementations of different control behaviors, including some representative examples of the use of continuations.

5.1 The Producer–Consumer Problem

The producer–consumer problem is the most paradigmatic example of the use of coroutines and constitutes a recurrent pattern in several scenarios. This problem involves the interaction of two independent computations: one that *produces* a sequence of items and one that *consumes* them, one at a time. Classical illustrations of this type of interaction use a pair of symmetric coroutines, with control being explicitly transfered back and forth between the producer and the consumer. However, asymmetric coroutines provide a simpler and more structured solution: we can implement the consumer as a conventional function that *resumes* the producer (an asymmetric coroutine) when the the next item is required[4].

A convenient extension of the producer–consumer structure is that of a *pipeline*, i.e., a producer–consumer chain consisting of an initial producer, one or more *filters* that perform some transformation on the transfered items, and a final consumer. Asymmetric coroutines provide a succinct implementation of pipelines too. A filter behaves both as a consumer and a producer, and can be implemented by a coroutine that resumes its antecessor to get a new value and yields the transformed value to its invoker (the next consumer in the chain). An implementation of a filter is shown in Figure 6. In a single statement we can create a pipeline by connecting the desired components and activating the final consumer:

```
consumer(filter(producer()))
```

5.2 Generators

A *generator* is a control abstraction that produces a sequence of values, returning a new value to its caller for each invocation. Actually, generators are nothing more than a particular instance of the producer–consumer problem, with the generator behaving as the producer.

[4]This is an example of a *consumer-driven* design. When appropriate, a *producer-driven* solution can also be developed.

```
function filter(ant)
  return coroutine.wrap(function()
                          while true do
                            -- resume antecessor to obtain value
                            local x = ant()
                            -- yield transformed value
                            coroutine.yield(f(x))
                          end
                        end)
end
```

Figure 6: Implementing a filter with asymmetric coroutines

A typical use of generators is to implement *iterators*, a related control abstraction that allows traversing a data structure independently of its internal implementation. Besides the capability of keeping state, the possibility of exchanging data when transferring control makes asymmetric coroutines a very convenient facility for implementing iterators. A classical example of an iterator implemented with Lua coroutines was shown in Section 3.3.2. However, the usefulness of generators is not restricted to implementing data-structure iterators. The next section provides an example of the use of generators in a quite different scenario.

5.3 Goal-oriented programming

Goal-oriented programming, as implemented in pattern-matching [Griswold and Griswold 1983] and also in Prolog-like queries [Clocksin and Mellish 1981] involves solving a problem or *goal* that is either a *primitive* goal or a *disjunction* of alternative goals. These alternative goals may be, in turn, *conjunctions* of subgoals that must be satisfied in succession, each of them contributing a partial outcome to the final result. In pattern-matching problems, matching string literals are primitive goals, alternative patterns are disjunctions of goals, and sequences of patterns are conjunctions of subgoals. In Prolog, the unification process is an example of a primitive goal, a relation constitutes a disjunction, and rules are conjunctions. In this context, solving a problem typically requires the implementation of a backtracking mechanism that successively tries each alternative until an adequate result is found.

Some implementations of Prolog-style backtracking in terms of continuations (e.g., [Haynes 1987]) use multi-shot *success* continuations to produce values, and are used as examples of scenarios where one-shot continuations cannot be used [Bruggeman et al. 1996]. However, this type of control behavior can be easily implemented with full asymmetric coroutines used as

```
-- matching a string literal (primitive goal)
function prim(str)
  return function(S, pos)
           local len = string.len(str)
           if string.sub(S, pos, pos+len-1) == str then
             coroutine.yield(pos+len)
           end
         end
end

-- alternative patterns (disjunction)
function alt(patt1, patt2)
  return function(S, pos)
           patt1(S, pos)
           patt2(S, pos)
         end
end

-- sequence of sub-patterns (conjunction)
function seq(patt1, patt2)
  return function(S, pos)
           local btpoint = coroutine.wrap(function()
                                            patt1(S, pos)
                                          end)
           for npos in btpoint do patt2(S, npos) end
         end
end
```

Figure 7: Goal-oriented programming: pattern matching

generators[5]. Wrapping a goal in a coroutine allows a backtracker (a simple loop) to successively retry (*resume*) the goal until an adequate result is found. A primitive goal can be defined as a function that *yields* a result at each invocation. 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.

As an example, let us consider a pattern-matching problem. Our goal is to match a string `S` with a pattern `patt`, which can be expressed by combining subgoals that represent alternative matchings or sequences of sub-patterns. An example of such a pattern is

```
("abc"|"de")."x"
```

[5]This style of backtracking can also be implemented with one-shot partial continuations, as shown by Sitaram [1993], or even with restricted forms of coroutines, such as Icon generators.

The implementation of pattern matching is shown in Figure 7. Each pattern function receives the subject string and a starting position. For each successful matching, it yields the next position to be checked. When it cannot find more matchings, it returns `nil`. Our primitive goal corresponds to matching a substring of `S` with a string literal. Function `prim` implements this goal; it receives as argument a string value and returns a function that tries to match it with a substring of `S` starting at the given position. If the goal succeeds, the position in `S` that immediately follows the match is yielded. Function `prim` uses two auxiliary functions from Lua's string library: `string.len`, which returns the length of a string, and `string.sub`, which returns a substring starting and ending at the given positions. Alternative patterns for a substring correspond to a disjunction of goals. They are implemented by function `alt`, which receives as arguments the two alternative goals and returns a function that tries to find a match in `S` by invoking these goals. If a successful match is found, the new position yielded by the invoked goal goes directly to the function's caller.

Matching a substring with a sequence of patterns corresponds to a conjunction of subgoals, which is implemented by function `seq`. The resulting pattern function creates an auxiliary coroutine (`btpoint`) to iterate on the first subgoal. Each successful match obtained by invoking this subgoal results in a new position in `S` where the second subgoal is to be satisfied. If a successful match for the second subgoal is found, the new position yielded by it goes directly to the function's caller.

Using the functions just described, the pattern `("abc"|"de")."x"` can be defined as follows:

```
patt = seq(alt(prim("abc"), prim("de")), prim("x"))
```

Finally, function `match` verifies if string `S` matches this pattern:

```
function match(S, patt)
  local len = string.len(S)
  local m = coroutine.wrap(function() patt(S, 1) end)
  for pos in m do
    if pos == len + 1 then
      return true
    end
  end
  return false
end
```

5.4 Cooperative Multitasking

One of the most obvious uses of coroutines is to implement multitasking. However, due mainly to the wide adoption of *multithreading* in modern mainstream languages, this suitable use of coroutines is currently disregarded.

```
-- list of "live" tasks
tasks = {}

-- create a task
function create_task(f)
  local co = coroutine.wrap(function() f(); return "ended" end)
  table.insert(tasks, co)
end

-- task dispatcher
function dispatcher()
  while true do
    local n = table.getn(tasks)
    if n == 0 then break end  -- no more tasks to run
    for i = 1,n do
      local status = tasks[i]()
      if status = "ended" then
        table.remove(tasks, i)
        break
      end
    end
  end
end
```

Figure 8: Implementing Cooperative Multitasking

A language with coroutines does not require additional concurrency constructs: just like a thread, a coroutine represents a unit of execution with its private, local data while sharing global data and other resources with other coroutines. But 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.

Preemptive scheduling and the consequent need for complex synchronization mechanisms make developing a correct multithreading application a difficult task. In some contexts, such as operating systems and real-time applications, timely responses are essential, and, therefore, preemption is unavoidable. However, the timing requirements for most concurrent applications are not critical. Moreover, most thread implementations do not provide any real timing guarantees. Application developers, also, have usually little or no experience in concurrent programming. In this scenario, a cooperative multitasking environment, which eliminates conflicts due to race conditions and minimizes the need for synchronization, seems much more appropriate.

The implementation of cooperative multitasking in terms of full asym-

metric coroutines is straightforward, as illustrated in Figure 8. Concurrent tasks are modeled by coroutines; when a new task is created, it is inserted in a list of *live* tasks. A simple task dispatcher is implemented by a loop that iterates on this list, resuming the live tasks and removing the ones that have finished their work (this condition is signalled by a predefined value returned by the coroutine main function). 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. For many concurrent applications, this is an unacceptable behavior. However, this situation is easily avoided by providing auxiliary functions that initiate an I/O operation and suspend the active coroutine when the operation cannot be immediately completed. Ierusalimschy [2003] shows an example of a concurrent application that uses Lua coroutines and includes non-blocking facilities.

Currently, there is some renewal of interest in cooperative multitasking as an alternative to multithreading [Adya et al. 2002; Behren et al. 2003]. However, the concurrent constructs that support cooperative multitasking in the proposed environments are usually provided by libraries or system resources like Window's *fibers* [Richter 1997]. Interestingly, although the description of the concurrency mechanisms employed in these environments is no more than a description of coroutines plus a dispatcher, the term coroutine is not even mentioned.

5.5 Exception Handling

A language that provides an exception handling mechanism typically implements two basic primitives: `try` and `raise` [Friedman et al. 2001]. The try primitive gets two expressions: a body and and an exception handler. When the body returns normally, its returned value becomes the value of `try`, and the exception handler is ignored. If the body encounters an exceptional condition, it raises an *exception* that is immediately sent to the handler; in this case, any unevaluated portion of the body is ignored. An exception handler can either return a value, which becomes the value of the associated `try`, or it can raise another exception, which is sent to the next dynamically enclosing handler.

A language with full asymmetric coroutines can easily support exception handling. A try primitive can be implemented as a function that gets two function values (`body` and `handler`) and executes `body` in a coroutine. A raise primitive is merely a function that *yields* an exception.

```
-- save original version of coroutine.wrap
local wrap = coroutine.wrap

-- redefine coroutine.wrap
function coroutine.wrap(tag, f)

   -- create a "tagged" coroutine
   local co = wrap(function(v) return tag, f(v) end)
   return function(v)
            local rtag, ret = co(v) -- resume coroutine
            while (rtag ~= tag) do

              -- reactivate outer coroutine if tags do not match
              v = coroutine.yield(rtag, ret)

              -- reinstate inner coroutine
              tag, ret = co(v)
            end

            -- tags match
            return ret
          end
end
```

Figure 9: Avoiding Interference Between Control Actions

5.6 Avoiding Interference Between Control Actions

When asymmetric coroutines implement different control structures in a single program and these structures are nested, it may be necessary to avoid interferences between control actions. An undesirable interference may arise, for instance, when an iterator is used within a `try` structure and it raises an exception.

We can avoid simple interferences if we identify pairs of control operations by associating each control structure with a different *tag* (a string, for instance) and implement a new version of function `wrap` that supports tags, as shown in Figure 9. In addition, function `coroutine.yield` takes a required first argument (a tag) in order to allow us to match yield–resume pairs. Notice that the basic idea of this solution is similar to that used for matching a subcomputation with its correspondent controller in our implementation of subcontinuations (see Section 4.3).

While avoiding simple interferences is trivial either with coroutines or continuations, in the presence of concurrency interferences between control actions, and error handling in general, can be hard to tackle [Gasbichler et al. 2003]. However, due to the compositional nature of asymmetric coroutines, handling errors does not present difficulties when concurrency is also

implemented with asymmetric coroutines.

6 Conclusions

After a period of intense investment, from the middle 1960s to the early 1980s, the research interest in coroutines as a general control abstraction virtually stopped. Besides the absence of a precise definition of the concept, which led to considerably different implementations of coroutine facilities, the other factors that greatly contributed to the discard of this interesting construct were the introduction of first-class continuations (and the general belief that they were far more expressive than coroutines), and the adoption of threads as a "standard" concurrent construct.

We now observe a renewal of interest in coroutines, notably in two different scenarios. The first corresponds to research efforts that explore the advantages of cooperative task management as an alternative to multithreading. In this scenario, some forms of coroutines are provided by libraries, or system resources, and are solely used as concurrent constructs. Another resurgence of coroutines is in the context of scripting languages, such as Python and Perl. In this case, restricted forms of coroutines support the implementation of simple iterators and generators, but are not powerful enough to constitute a general control abstraction; in particular, they cannot be used as a concurrent construct.

In this paper we argued in favor of the revival of *full* asymmetric coroutines as a convenient general control construct, which can replace both one-shot continuations and multithreading with a single, and simpler, concept. In order to support this proposition, we provided the contributions described next.

To fulfill the need of an adequate definition of the concept of a coroutine, we proposed a classification of coroutines based on three main issues: whether coroutines are symmetric or asymmetric, whether they are first-class objects, and whether they are stackful constructs. We discussed the influence of each of these issues on the expressive power of a coroutine facility, and introduced the concept of *full* coroutines as first-class, stackful objects. We also discussed the advantages of full *asymmetric* coroutines versus full *symmetric* coroutines, which are equivalent in power, but not in ease of use.

Next we provided a precise definition of a full asymmetric coroutine construct, supported by the development of an operational semantics for this mechanism. We then demonstrated that full asymmetric coroutines can provide not only symmetric coroutines but also one-shot continuations and one-shot partial continuations, and discussed the similarities between one-shot continuations and full symmetric coroutines, and between one-shot partial continuations and full asymmetric coroutines. We also showed that,

although these constructs have equivalent power, they may not be equivalent in terms of efficiency.

Finally, we provided a collection of programming examples that illustrate the use of full asymmetric coroutines to support concise and elegant implementations of several useful control behaviors, including some of the most relevant examples of the use of continuations.

References

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 Annual Technical Conference*. USENIX, Monterey, CA.

Behren, R., Condit, J., and Brewer, E. 2003. Why Events are a Bad Idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. Lihue, HI.

Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. 1980. *Simula Begin*. Studentlitteratur, Sweden.

Bruggeman, C., Waddell, O., and Dybvig, R. 1996. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN'96 Conf. on Programming Language Design and Implementation (PLDI)*. ACM, Philadelphia, PA, 99–107. SIGPLAN Notices 31(5).

Clocksin, W. and Mellish, C. 1981. *Programming in Prolog*. Springer-Verlag.

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

Conway, M. 1963. Design of a separable transition-diagram compiler. *Communications of the ACM 6,* 7 (July), 396–408.

Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. 1972. Hierarchical program structures. In *Structured Programming*, Second ed. Academic Press, London, England.

Danvy, O. and Filinski, A. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. ACM, Nice, France, 151–160.

Dybvig, R. and Hieb, R. 1989. Engines from continuations. *Computer Languages 14,* 2, 109–123.

Felleisen, M. 1985. Transliterating Prolog into Scheme. Technical Report 182, Indiana University.

Felleisen, M. 1988. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages POPL'88*. ACM, San Diego, CA, 180–190.

Felleisen, M. 1990. On the expressive power of programming languages. In *Proceedings of 3rd European Symposium on Programming ESOP'90*. Copenhagen, Denmark, 134–151.

Felleisen, M. and Friedman, D. 1986. Control operators, the secd-machine, and the λ-calculus. In *Formal Description of Programming Concepts-III*, M. Wirsing, Ed. North-Holland, 193–217.

Friedman, D., Haynes, C., and Kohlbecker, E. 1984. Programming with continuations. In *Program Transformation and Programming Environments*, P. Pepper, Ed. Springer-Verlag.

Friedman, D., Wand, M., and Haynes, C. 2001. *Essentials of Programming Languages*, Second ed. MIT Press, London, England.

Gasbichler, M., Knauel, E., and Sperber, M. 2003. How to add threads to a sequential language without getting tangled up. In *Proceedings of the 4th Workshop on Scheme and Functional Programming*. Cambridge, MA, 30–47.

Griswold, R. and Griswold, M. 1983. *The Icon Programming Language*. Prentice-Hall, New Jersey, NJ.

Harper, R., Duba, B., Harper, R., and MacQueen, D. 1991. Typing first-class continuations in ML. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages POPL'91*. ACM, Orlando, FL, 163–173.

Haynes, C. T. 1987. Logic continuations. *J. Logic Programming 4*, 157–176.

Haynes, C. T., Friedman, D., and Wand, M. 1986. Obtaining coroutines with continuations. *Computer Languages 11,* 3/4, 143–153.

Hieb, R., Dybvig, R., and Anderson III, C. W. 1994. Subcontinuations. *Lisp and Symbolic Computation 7,* 1, 83–110.

Hieb, R., Dybvig, R., and Bruggeman, C. 1990. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation (PLDI)*. ACM, White Plains, NY, 66–77. SIGPLAN Notices 25(6).

Ierusalimschy, R. 2003. *Programming in Lua*. Lua.org, ISBN 85-903798-1-7, Rio de Janeiro, Brazil.

Ierusalimschy, R., Figueiredo, L., and Celes, W. 1996. Lua — an extensible extension language. *Software: Practice & Experience 26,* 6 (June), 635–652.

Johnson, G. and Duggan, D. 1988. Stores and partial continuations as first-class objects in a language and its environment. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages POPL'88*. ACM, San Diego, CA.

Kelsey, R., Clinger, W., and Rees, J. 1998. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33,* 9 (Sept.), 26–76.

Knuth, D. E. 1968. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*. Addison-Wesley, Reading, MA.

Kumar, S., Bruggeman, C., and Dybvig, R. 1998. Threads yield continuations. *Lisp and Symbolic Computation 10,* 3, 223–236.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1977. Abstraction mechanisms in CLU. *Communications of the ACM 20,* 8 (Aug.), 564–576.

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

Moody, K. and Richards, M. 1980. A coroutine mechanism for BCPL. *Software: Practice & Experience 10,* 10 (Oct.), 765–771.

Moura, A., Rodriguez, N., and Ierusalimschy, R. 2004. Coroutines in lua. In *8th Brazilian Symposium of Progamming Languages (SBLP)*. SBC, Niteroi, RJ, Brazil.

Murer, S., Omohundro, S., Stoutamire, D., and Szyperski, C. 1996. Iteration abstraction in Sather. *ACM Transactions on Progamming Languages and Systems 18,* 1 (Jan.), 1–15.

Pauli, W. and Soffa, M. L. 1980. Coroutine behaviour and implementation. *Software: Practice & Experience 10,* 3 (Mar.), 189–204.

Queinnec, C. 1993. A library of highr-level control operators. *ACM SIGPLAN Lisp Pointers 6,* 4 (Oct.), 11–26.

Queinnec, C. and Serpette, B. 1991. A dynamic extent control operator for partial continuations. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages POPL'91*. ACM, Orlando, FL, 174–184.

Richter, J. 1997. *Advanced Windows*, Third ed. Microsoft Press, Redmond, WA.

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

SITARAM, D. 1993. Handling control. In *Proceedings of the ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation (PLDI)*. ACM, Albuquerque, NM. SIGPLAN Notices 28(6).

SITARAM, D. 1994. Models of control and their implications for progamming language design. Ph.D. thesis, Rice University.

TISMER, C. 2000. Continuations and Stackless Python. In *Proceedings of the 8th International Python Conference*. Arlington, VA.

WAND, M. 1980. Continuation-based multiprocessing. In *Proceedings of the 1980 Lisp Conference*. ACM, Stanford, CA, 19–28.

WIRTH, N. 1985. *Programming in Modula-2*, Third, corrected ed. Springer-Verlag.