Revisiting Coroutines

ANA LÚCIA DE MOURA and ROBERTO IERUSALIMSCHY
Catholic University of Rio de Janeiro (PUC–Rio)

This article advocates the revival of coroutines as a convenient 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 delimited continuations. We also show that full asymmetric coroutines and one-shot delimited continuations have many similarities, and therefore present comparable benefits. Nevertheless, coroutines are easier implemented and understood, especially in the realm of procedural languages.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures, coroutines*

General Terms: Design, Languages

Additional Key Words and Phrases: Continuations, generators, multitasking

ACM Reference Format:
de Moura, A. L. and Ierusalimschy, R. 2009. Revisiting coroutines. ACM Trans. Program. Lang. Syst. 31, 2, Article 6 (February 2009), 31 pages. DOI = 10.1145/1462166.1462167 http://doi.acm.org/10.1145/1462166.1462167

1. INTRODUCTION

The concept of coroutines was introduced in the early 1960's 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 different contexts, including simulation, artificial intelligence, concurrent programming, text processing, and various kinds of data-structure manipulation [Knuth 1968; Marlin 1980; Pauli

This work was partially supported by grants from CNPq (the Brazilian Research Council), Microsoft Research, and Tecgraf (Computer Graphics Technology Group, PUC–Rio).
Authors' addresses: A. L. de Moura, R. Ierusalimschy, Catholic University of Rio de Janeiro (PUC–Rio), Rua M. S. Vicente 225, Rio de Janeiro, RJ; 22451-900, Brazil; email: ana@rnp.br, roberto@inf.puc-rio.br.

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 lack of a uniform view of this concept, which was never precisely defined. Marlin's doctoral thesis [Marlin 1980], widely acknowledged as a reference for this mechanism, summarizes 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 different kinds of 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 implemented as a *stackful* construct, that is, whether it is able to suspend its execution from within nested calls.

Depending on the intended use for the coroutine mechanism, particular solutions for these 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 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 [Kelsey et al. 1998] also 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, 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.

[1]We will define our concept of expressiveness in Section 5.

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 more recent 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 article is to advocate 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 (i.e., when a full coroutine mechanism is implemented) coroutines have power equivalent to one-shot continuations and one-shot delimited continuations.[2] We also demonstrate that symmetric and asymmetric coroutines have equivalent expressive power. However, since asymmetric coroutines are easier to manage and understand, and support more structured applications, we specifically defend full *asymmetric* coroutines as a convenient construct for language extensibility.

The remainder of this article 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. Section 4 contains a collection of programming examples that use full asymmetric coroutines to implement some useful control behaviors. In Section 5 we show that full asymmetric coroutines can provide not only symmetric coroutine facilities but also one-shot continuations and one-shot delimited continuations. Section 6 summarizes the work and presents some final remarks.

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, 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 coroutine facilities concerns the provided control-transfer operations and distinguishes the concepts of *symmetric* and *asymmetric* coroutines. Symmetric coroutine facilities provide a single control-transfer

[2]The usual implementation of coroutines using continuations (e.g., Springer and Friedman [1989]) uses continuations in a one-shot manner. Similarly, the implementation of coroutines using delimited continuations presented by Sitaram [1994] also uses continuations in a one-shot manner. So, here we are concerned only with the reverse direction: how to implement one-shot continuations and delimited continuations with coroutines.

operation that allows coroutines to explicitly pass control among themselves. Asymmetric coroutine mechanisms (more commonly denoted as *semisymmetric* 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 that 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 are not equally powerful. However, we demonstrate in Section 5 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 transferring control among themselves may require considerable effort from a programmer. On the other hand, since asymmetric coroutines always transfer control back to their invokers, control sequencing is much simpler to manage and understand. The composable behavior of asymmetric coroutines also provides support for concise implementations of several useful control behaviors, including generators, goal-oriented programming, and multitasking environments, as we will show in Section 4. Although implementing these control behaviors with symmetric coroutines is also possible, it complicates considerably the structure of 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 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 [Griswold and Griswold 1983]. 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 and Modula-2.

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 a `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 nonstackful 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 a hierarchy of auxiliary generators that yield in succession until the original invocation point is reached.

Nonstackful coroutines are also not powerful enough to support general-purpose multitasking environments. Most programs perform I/O through auxiliary libraries which usually require the suspension of a task during the execution of some I/O operation to avoid blocking the whole program (e.g., when no data is available). With nonstackful coroutines, a task cannot yield within those libraries. Therefore, either those programs will block until the operation completes or they must be restructured to avoid doing I/O inside library functions.

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 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. Moreover, as we discussed in Moura et al. [2004], asymmetric coroutines can be used even in programs that call external procedures developed in languages that do not provide coroutine support.[3] Therefore, we believe that full asymmetric coroutine mechanisms provide a more convenient control abstraction than symmetric coroutine 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.[4]

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

[3]This statement seems to contradict our claims that both forms of coroutines have equivalent expressive power. However, the notion of expressive power is limited to constructions *within* the language and therefore does not apply to interlanguage constructions, which are not part of our formal model.

[4]This semantics is a modified version of the semantics we presented in Moura et al. [2004].

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 beginning of 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.

The coroutine operators 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, this 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. Our approach is similar to the operational semantics of *subcontinuations*, described by Hieb et al. [1994]. We start with a 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 labels (l, a set of constant values), variables (x), function definitions (abstractions), function calls (applications), assignments, conditionals, an equality operator for labels, and a **nil** value.

$$e \;\rightarrow\; l \mid x \mid \lambda x \cdot e \mid e\, e \mid x := e \mid \mathbf{if}\, e\; \mathbf{then}\, e\; \mathbf{else}\, e \mid e = e \mid \mathbf{nil}$$

Expressions that denote values (v) are labels, functions, and **nil**.

$$v \;\rightarrow\; l \mid \lambda x \cdot e \mid \mathbf{nil}$$

As usual, $FV(e)$ is the set of free variables in e; we also define $LB(e)$ as the set of labels in e.

A store θ, mapping variables and labels to values, is included in the definition of the core language to allow side-effects.[5]

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

[5]The core language does not provide a means to bind a value to a label. Its extensions, however, will include and use that facility.

We extend the definition of FV and of LB to denote the free variables and the labels in a store.

$$FV(\theta) = \bigcup_{x \in \mathrm{dom}(\theta)} FV(\theta(x))$$
$$LB(\theta) = \bigcup_{x \in \mathrm{dom}(\theta)} LB(\theta(x))$$

The evaluation of the core language is defined by a set of rewrite rules that are applied to states (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 our core language specify a right-to-left evaluation[6] of applications.

$$C \;\to\; \square \mid e\,C \mid C\,v \mid x := C \mid \mathbf{if}\,C\,\mathbf{then}\,e\,\mathbf{else}\,e \mid e = C \mid C = v$$

The rewrite rules for evaluating the core language are given next.

$$\langle C[x],\ \theta\rangle \;\Rightarrow\; \langle C[\theta(x)],\ \theta\rangle \tag{1}$$
$$\langle C[(\lambda x \cdot e)v],\ \theta\rangle \;\Rightarrow\; \langle C[e[z/x]],\ \theta[z \leftarrow v]\rangle,\quad z \notin (FV(\theta) \cup FV(C[(\lambda x \cdot e)v])) \tag{2}$$
$$\langle C[x := v],\ \theta\rangle \;\Rightarrow\; \langle C[v],\ \theta[x \leftarrow v]\rangle,\ x \in \mathrm{dom}(\theta) \tag{3}$$
$$\langle C[\mathbf{if}\,v\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2],\ \theta\rangle \;\Rightarrow\; \langle C[e_1],\ \theta\rangle, v \neq \mathbf{nil} \tag{4}$$
$$\langle C[\mathbf{if}\,\mathbf{nil}\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2],\ \theta\rangle \;\Rightarrow\; \langle C[e_2],\ \theta\rangle \tag{5}$$
$$\langle C[l = l],\ \theta\rangle \;\Rightarrow\; \langle C[l],\ \theta\rangle \tag{6}$$
$$\langle C[l_1 = l_2],\ \theta\rangle \;\Rightarrow\; \langle C[\mathbf{nil}],\ \theta\rangle, l_1 \neq l_2 \tag{7}$$

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, variable renaming is made to guarantee the use of a fresh variable z. 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). Rules 4 and 5 describe conditionals which test whether the condition is **nil** in order to choose a branch. The last two rules describe the equality operator.

We say that an expression (or a *program*) e *results* in a value v, denoted as $e \Downarrow v$, when $\langle e,\ \theta_0\rangle \overset{*}{\Rightarrow} \langle v,\ \theta\rangle$, where θ_0 is the empty store and θ is any arbitrary store. As usual, $\overset{*}{\Rightarrow}$ is the reflexive-transitive closure of $\Rightarrow$.

When needed, we will use the usual syntactic sugar $\mathbf{let}\,x = e_1\ \mathbf{in}\,e_2$ meaning $(\lambda x \cdot e_2)e_1$ (where x may occur free in e_2); and $e_1; e_2$ meaning $(\lambda x \cdot e_2)e_1$ for some x not free in e_2. We assume that **let** has a lower precedence than the semicolon and that the semicolon has a lower precedence than the other operations; for instance, $\mathbf{let}\,x = e_1\ \mathbf{in}\,x := e_2; e_3$ binds as $\mathbf{let}\,x = e_1\ \mathbf{in}\,((x := e_2); e_3)$.

In order to incorporate asymmetric coroutines into the language, we extend the set of expressions with *labeled expressions* (l: e), plus the coroutine

[6] Later it will become clear why we use right-to-left evaluation instead of the more usual left-to-right order.

operators.

$$e \;\rightarrow\; \ldots \mid l\colon e \mid \textbf{create}\, e \mid \textbf{resume}\, e\; e \mid \textbf{yield}\, e$$

In our extended language, which we will call λ_a, 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. The precedence of the prefix operator l: is lower than that of the semicolon.

The definition of evaluation contexts must include the new expressions. In this new definition we specify a right-to-left evaluation for the operator *resume*.

$$C \;\rightarrow\; \ldots \mid \textbf{create}\, C \mid \textbf{resume}\, e\; C \mid \textbf{resume}\, C\; v \mid \textbf{yield}\, C \mid l\colon C$$

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 ($l\colon C$). 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.

$$\langle C[\textbf{create}\, v],\, \theta\rangle \;\Rightarrow\; \langle C[l\,],\, \theta[l\, \leftarrow v]\rangle,\; l \not\in (LB(\theta) \cup LB(C[\textbf{create}\, v])) \quad (8)$$

$$\langle C[\textbf{resume}\, l\; v],\, \theta\rangle \;\Rightarrow\; \langle C[l\colon (\theta(l\,)\, v)],\, \theta[l\, \leftarrow \textbf{nil}]\rangle \quad (9)$$

$$\langle C_1[l\colon C'_2[\textbf{yield}\, v]],\, \theta\rangle \;\Rightarrow\; \langle C_1[v],\, \theta[l\, \leftarrow \lambda x \cdot C'_2[x]]\rangle \quad (10)$$

$$\langle C[l\colon v],\, \theta\rangle \;\Rightarrow\; \langle C[v],\, \theta\rangle \quad (11)$$

Rule 8 describes the action of creating a coroutine. It creates a new label to represent the coroutine and stores a mapping from this label to the coroutine main function.

Rule 9 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 **nil**.

Rule 10 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 corresponding 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 **nil**, established when the coroutine was resumed, prevents the reactivation of a dead coroutine.

4. PROGRAMMING WITH FULL ASYMMETRIC COROUTINES

This section provides some programming examples that illustrate the use of full asymmetric coroutines as a general control construct. We begin by presenting a full asymmetric coroutine facility provided by the general-purpose programming language Lua. This facility is then used to implement different useful control behaviors, including some representative examples of the use of continuations.

4.1 An Example of a Full Asymmetric Coroutine Facility

As we mentioned in Section 2.3, the incorporation of asymmetric coroutine facilities is a growing trend in the context of modern scripting languages such as Python and Perl. However, the coroutine mechanisms provided by these languages are typically intended for supporting specific constructs, and do not implement our concept of full coroutines. In particular, they do not provide stackful coroutines.

Lua [Ierusalimschy et al. 1996; Ierusalimschy 2003] follows a different approach. Since its version 5.0, Lua provides full asymmetric coroutines intended for use as a general control abstraction.

4.1.1 *An Overview of Lua.* Lua is a lightweight scripting language that supports general procedural programming with data description facilities. It is dynamically typed, lexically scoped, and has automatic memory management.

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
```

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.

Function parameters work exactly as local variables, initialized with the values of the actual arguments provided in the function call. Lua adjusts the number of actual arguments to the number of parameters: Extra arguments are thrown away, extra parameters are given a `nil` value.

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 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.

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

function inorder_iterator(tree)
  return coroutine.wrap(function()
                          inorder(tree)
                          return nil
                        end)
end
```

Fig. 1. A binary tree iterator implemented with Lua coroutines.

4.1.2 *Lua Coroutines.* Except for some additional features supported by the language, Lua coroutine facilities follow the semantics of full asymmetric coroutines described in Section 3. Moura et al. [2004] provide a detailed description of these facilities; we present here only the facilities that we use in our programming examples.

As in most Lua libraries, Lua coroutine operations are packed in a global table (table `coroutine`). Function `coroutine.wrap` creates a new coroutine. It receives as argument a Lua function that represents the main body of the coroutine and returns a function that, when called, *resumes* that coroutine.[7] The semantics of the Lua function `wrap` can be easily defined in terms of the operators **create** and **resume** of the language λ_a.

$$wrap = \lambda f \cdot \mathbf{let}\, l = \mathbf{create}\ f\ \ \mathbf{in}\, \lambda x \cdot \mathbf{resume}\ l\ x$$

Function `coroutine.yield` basically follows the semantics of the operator **yield**. It suspends the execution of the active coroutine, and returns control to that coroutine's last activation point.

4.2 Implementing Generators

A *generator* is a control abstraction that produces a sequence of values, returning a new value to its caller for each invocation. Besides the capability of keeping state, the possibility of exchanging data when transferring control makes asymmetric coroutines a very convenient facility for implementing generators.

A typical use of generators is to implement *iterators*, a related control abstraction that allows traversing a data structure. The Lua code shown in Figure 1 implements a classical example: an iterator that traverses a binary tree in in-order. In this example, tree nodes are represented by Lua tables

[7]Lua also offers a more primitive operation for creating coroutines, called `coroutine.create`. This operation returns a coroutine object that allows other operations over the coroutine, such as inspecting its stack, its status, etc. When we do not need these facilities, the higher-level `wrap` is simpler to use.

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

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

Fig. 2. Merging two binary trees.

containing three fields: `key`, `left`, and `right`. Field `key` stores the node value (a number); fields `left` and `right` contain references to the node's respective children.

Function `inorder_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 (`inorder`) that yields the produced value directly to the iterator's caller. The end of a traversal is signaled 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 corresponding iterator for obtaining its next element, continuing until the elements in both trees are exhausted.

However useful, implementing data-structure iterators is not the only application for generators. The next section provides an example of the use of generators in a quite different scenario.

4.3 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 authors (e.g., Bruggeman et al. [1996]) cite some implementations of Prolog-style backtracking (e.g., Haynes [1987]) as a scenario that demands multishot continuations. This implementation is based on a well-known model of backtracking computation—the *two-continuation model*—that uses two types of continuations: a multishot *success* continuation, which consumes a candidate answer, and a one-shot *failure* continuation that is invoked to get another answer [Wand and Vaillancourt 2004].

However, this type of control behavior can be easily implemented with full asymmetric coroutines used as generators.[8] 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 subpatterns. An example of such a pattern is as follows.

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

Figure 3 shows our implementation of pattern-matching with Lua coroutines. 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 that function's caller.

Matching a substring with a sequence of patterns corresponds to a conjunction of subgoals, implemented by function `seq`. The resulting pattern function creates an auxiliary coroutine (`btpoint`) to iterate on the first subgoal. Each

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

```
-- 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
```

Fig. 3. Goal-oriented programming: pattern matching.

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
```

Danvy and Filinski [1990] present two different implementations of pattern-matching, one using delimited continuations in the form of shift/reset primitives, and the other using explicit continuation passing style.

Both implementations need multishot continuations for the success case. Coroutines support only one-shot continuations, so our implementation follows a different approach. If we observe the CPS implementation, it is easy to see that the only places where it creates continuations are the conjunction (concatenation) operation and the main match function. In our implementation, these operations use an explicit loop to create distinct continuations for each success match, thus avoiding invoking the same continuation twice.

4.4 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.

A language with coroutines does not require additional concurrency constructs. Like threads and engines [Haynes and Friedman 1987a], coroutines embody independent computations that can be suspended and later restarted from the point of suspension. However, while engines and most implementations of threads provide preemption-based multitasking, coroutines provide a concurrency model which is essentially cooperative: A coroutine must voluntarily release control to allow other coroutines to proceed.

In a cooperative multitasking environment, the interleaving of concurrent tasks is deterministic, and race conditions do not occur. Coordinating access to shared resources is a simple task, and the need for synchronization mechanisms is minimized. Preemptive scheduling needs more complex synchronization mechanisms, making the development of correct multithreading applications 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, differently from operating system developers, application developers usually have little or no experience in concurrent programming. In this scenario, a cooperative multitasking environment seems more appropriate.

Cooperative multitasking can involve occasional fairness problems when concurrent tasks execute time-consuming operations. In nonpreemptive operating systems, where the concurrent tasks are typically noncollaborative independent programs, fairness problems can be rather difficult to solve. In this context, when a program monopolizes processing resources, it impacts the progress of other programs, not its own. As a consequence of this behavior, program developers will neither be aware of the need to insert yield statements in time-consuming operations, nor motivated to do so. Moreover, performance problems and starvation conditions are not easily reproduced, and there is virtually no way for the programmer to check where suspension requests can be adequately inserted.

```
-- 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()
  local i = 1
  while true do
    if tasks[i] == nil then
      if tasks[1] == nil then break end
      i = 1
    end
    local status = tasks[i]()
    if status == "ended" then
      table.remove(tasks, i)
    else
      i = i + 1
    end
  end
end
```

Fig. 4. Implementing cooperative multitasking.

User-level multitasking presents a somewhat different scenario, where coroutines are part of the same program and collaborate to achieve a common goal. Since fairness problems are restricted to the collaborative environment, they are more easily identified and reproduced. Therefore, with cooperative user-level multitasking, fairness is not as big a problem as it is in a noncooperative scenario. Nevertheless, fairness is still a difficulty when programming cooperative multitasking applications.[9]

Implementing cooperative multitasking in terms of full asymmetric coroutines is straightforward, as illustrated in Figure 4. Concurrent tasks are modeled by coroutines; when a new task is created, it is inserted in a list of *live* tasks implemented by a Lua table. This table, initially empty, is created by the *constructor expression* `{}`.

A simple task dispatcher can be implemented by a loop that iterates on this list, resuming the live tasks and removing the ones that have finished their work. (Function `table.remove` removes the given item from a table, moving down the upper indices to close the gap.) The end of a task is signaled by a predefined value returned by the coroutine main function (the string `"ended"`). When a task suspends its execution (by calling function `coroutine.yield` with no arguments), a `nil` value is returned to the dispatcher.

[9]We may say that by removing preemption, we have easier correctness at the expense of harder fairness.

Besides fairness, another drawback of user-level 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. Nonetheless, 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 nonblocking facilities. That example makes use of a `select` operation when all coroutines are blocked, thus avoiding unnecessary polling.

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 most of 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 those environments is no more than a description of coroutines plus a dispatcher, the term *coroutine* is not even mentioned.

Fair Threads [Serrano et al. 2004] is an example of an interesting model of concurrent programming which combines preemptive *service* threads and cooperative *user* threads. In this model, user threads cooperate either explicitly (by means of a *yield* operation) or implicitly (by waiting for *signals*). A cooperation mechanism based on signaling conditions is not difficult to add to our coroutine scheduler.

5. EXPRESSING ALTERNATIVE CONTROL STRUCTURES

In the previous section, we provided some examples that illustrate the usefulness of full asymmetric coroutines as a general control construct. In this section we explore the expressive power of full asymmetric coroutines relative to other control abstractions. We will show that full asymmetric coroutines can implement not only symmetric coroutines, but also one-shot continuations and one-shot delimited continuations; therefore, they can provide any sort of control structure implemented by those constructs. In other words, asymmetric coroutines has the same expressive power of these other constructs.

Our notion of expressive power is as follows: Given two languages A and B with a common core, differing only in that one has a set of operators $\{a_1, \ldots, a_n\}$ and the other a set $\{b_1, \ldots, b_m\}$, we say that A has (at least) the same expressive power of B (or that A is as least as expressive as B) if there is a context C such that, if a program e results in v in language B, then the program $C[e]$ also results in v in language A.[10]

[10]Formally, we must also allow specific local transformations from the original expression in language B to the expression in language A: Any application of an operator b_i, which is a primitive operation in B, is changed to a regular function application in A.

5.1 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. Therefore, our model of symmetric coroutines needs only two basic operators: *create* and *transfer*. For convenience, it also provides another operator, *current*, that returns a reference to the running (current) coroutine.

Creating a symmetric coroutine is similar to creating an asymmetric coroutine: The operator *create* receives a procedural argument (the coroutine main body) and returns a reference to the new coroutine. The operator *transfer* saves the continuation point of the current coroutine and (re)activates the coroutine whose reference is passed as its first argument. The reactivated coroutine starts executing at its saved continuation point and runs until it transfers control to another coroutine, or until its main function terminates. The end of the main coroutine is the end of the program; the end of any other coroutine implicitly transfers the control back to the main coroutine.

Like our asymmetric coroutines, our symmetric coroutines can exchange data; when a coroutine transfers control, the second argument given to the operator *transfer* becomes the result value of the transfer operation which suspended the reactivated coroutine.

Let us formalize our model of symmetric coroutines. We do so by extending the core language introduced in Section 3.2. We will call this extended language λ_{sym}. We begin by extending the set of expressions with the symmetric coroutine operators.

$$\begin{array}{rcl} e & \rightarrow & l \mid x \mid \lambda x \cdot e \mid e\, e \mid x := e \mid \textbf{if}\, e\, \textbf{then}\, e\, \textbf{else}\, e \mid e = e \mid \textbf{nil} \mid \\ & & \textbf{create}\, e \mid \textbf{transfer}\, e\, e \mid \textbf{current} \end{array}$$

We also extend the definition of evaluation contexts to include the new expressions, specifying a right-to-left evaluation for the arguments of *transfer*.

$$\begin{array}{rcl} C & \rightarrow & \square \mid e\, C \mid C\, v \mid x := C \mid \textbf{if}\, C\, \textbf{then}\, e\, \textbf{else}\, e \mid C = e \mid v = C \mid \\ & & \textbf{create}\, C \mid \textbf{transfer}\, e\, C \mid \textbf{transfer}\, C\, v \end{array}$$

In our semantics for symmetric coroutines, rewrite rules are applied to expression-store-label triplets; the third element of this triplet represents the active coroutine. A distinguished label l_I identifies the main coroutine. The first rules of λ_{sym} are similar to rules 1–7 of the core language, except that they operate on triplets instead of pairs, never changing the active coroutine.

The new rules for the symmetric coroutine operations are as follows.

$$\langle C[\textbf{create}\, v],\, \theta,\, l_1\rangle \Rightarrow \langle C[l_2],\, \theta[l_2 \leftarrow v],\, l_1\rangle, \text{ where } l_2 \text{ is a fresh label} \quad (12)$$

$$\langle C[\textbf{transfer}\, l_2\, v],\, \theta,\, l_1\rangle \Rightarrow \langle \theta(l_2)\, v,\, \theta[l_2 \leftarrow \textbf{nil},\, l_1 \leftarrow \lambda x \cdot C[x]],\, l_2\rangle,\, l_1 \neq l_2 \quad (13)$$

$$\langle C[\textbf{transfer}\, l\ v],\, \theta,\, l\rangle \Rightarrow \langle C[v],\, \theta,\, l\rangle \quad (14)$$

$$\langle C[\textbf{current}],\, \theta,\, l\rangle \Rightarrow \langle C[l],\, \theta,\, l\rangle \quad (15)$$

$$\langle v,\, \theta,\, l\rangle \Rightarrow \langle \theta(l_I)\, v,\, \theta[l_I \leftarrow \textbf{nil}],\, l_I\rangle,\, l \neq l_I \quad (16)$$

Rule 12 describes the semantics of creating a symmetric coroutine; this operation is similar to the creation of an asymmetric coroutine, described in Section 3.2.

Rule 13 describes the transfer of control between symmetric coroutines. A transfer binds the current label l_1 to the current continuation C in the store, and gets the continuation of the coroutine to be (re)activated ($\theta'(l_2)$). This continuation is then invoked with the second argument passed to *transfer*. Rule 14 handles the particular case where a coroutine transfers control to itself; this operation simply returns the given value.

Rule 15 provides the semantics for the **current** primitive.

Rule 16 describes what happens when a coroutine ends. The continuation of the main coroutine is invoked with the value given by the ending coroutine. The end of the main coroutine ends the program, so there is no rule for that case. We say that $e \Downarrow v$ if $\langle e, \theta_0, l_I\rangle \overset{*}{\Rightarrow} \langle v, \theta, l_I\rangle$ for some store θ.

The implementation of symmetric coroutines on top of asymmetric facilities is not difficult. Symmetrical transfers of control between asymmetric coroutines can be 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. Following these guidelines, Figure 5 shows the implementation of symmetric coroutines in Lua. When function `transfer` is called for the first time (i.e., by the main coroutine), the dispatching loop is started. When `transfer` is invoked by an active coroutine, function `coroutine.yield` is called to reactivate the dispatcher. Global variable `current` always contains a reference to the running (current) coroutine. An auxiliary variable (`main`) represents the main coroutine (it has the same role as label l_I in the semantics of λ_{sym}.)

The following definition is a straightforward translation of `transfer` and `current` in Figure 5 to the language λ_a:[11]

```
let $current = \textbf{create}\ \lambda x \cdot x$ in
let $transfer =$
  let $main = current$ in
  let $next = main$ in
  let $disp = \textbf{nil}$ in
  $disp := \lambda val\,.$
    if $current = main$ then $val$ else $(next := main; C_d[\textbf{resume}\ current\ val])$;
  $\lambda co \cdot \lambda val\,.$
    if $current = main$ then $current := co; disp\ val$ else $(next := co; \textbf{yield}\ val)$
in $\cdots$
```

where the context C_d is defined as follows.

$$\textbf{let}\ val = \Box\ \textbf{in}\ current := next; disp\ val$$

[11]We use the technique of first declaring the variable *disp* and then defining its value to allow a recursive function.

```
-- creates a distinguished coroutine
current = coroutine.wrap(function (x) end)

local next = main
local main = current

create = coroutine.wrap

transfer = function(co, val)
  if current == main then
    current = co
    while current  = main -- dispatching loop
      next = main
      val = current(val)
      current = next
    end
    return val
  else
    next = co
    return coroutine.yield(val)
  end
end
```

Fig. 5. Implementing symmetric coroutines with Lua asymmetric coroutines.

To prove the correctness of this definition we must show that for any expression e, if $e \Downarrow v$ in λ_{sym}, then ($\textbf{let}\, current = \cdots$ $\textbf{in let}\, transfer = \cdots$ $\textbf{in}\, e) \Downarrow v$ in λ_a. Roughly, we can prove this by defining a mapping from states in λ_{sym} to states in λ_a, and then showing that the transition rules preserve the mapping. In Ierusalimschy and de Moura [2008] we present the complete proof.

For completeness, we will now show how to emulate λ_a programs on top of λ_{sym}, that is, how to implement **resume–yield** using **transfer**. A naive (but slightly wrong) implementation could be like the one given next.

```
let $yield = \textbf{nil}$ in
let $resume = \lambda co \cdot \lambda val$ .
         let $previous = \textbf{current}$ in
         let $oldyield = yield$ in
           $yield := \lambda val \cdot (yield := oldyield; \textbf{transfer}\ previous\ val)$;
           transfer $co\ val$
in $\cdots$
```

Function *yield* is initially **nil** because the main coroutine cannot yield. Function *resume* contains the bulk of the implementation. First it saves the current coroutine and the current value of *yield*. Then it redefines *yield* as a function that, when called, restores *yield* and transfers control back to the now-current coroutine. Finally, *resume* transfers control to the invoked coroutine.

The problem with this definition happens when a coroutine terminates its main function. Language λ_{sym} transfers control back to the main coroutine, but in λ_a control should return to the corresponding **resume**. To solve this problem, we insert in *resume* a new variable *test* which controls whether *yield* was properly called.

```
let $yield$ = nil in
let $resume = \lambda co \cdot \lambda val$ .
          let $previous$ = current in
          let $oldyield = yield$ in
          let $test = nil$ in
             $yield := \lambda val \cdot (yield := oldyield; test := \lambda x \cdot x;$ transfer $previous\ val)$;
             $test := \lambda val \cdot test\ (yield\ val)$;
             $test$ (transfer $co\ val$)
  in ···
```

When a coroutine yields, *test* is set to the identity function, so it has no effect and *resume* behaves as before. When a coroutine returns without yielding (i.e., its main function returns), *test* yields the returned value (and repeats the test when control eventually returns).

5.2 One-Shot Continuations

Although conventional first-class continuation mechanisms allow a continuation to be invoked multiple times, in virtually all their useful applications continuations can be invoked only once. Motivated by this fact, Bruggeman et al. [1996] implemented the concept of *one-shot* continuations [Haynes and Friedman 1987b], introducing the control operator `call/1cc`. One-shot continuations differ from multishot 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`).

We again will extend our core language to create the language λ_{c1cc}, which supports one-shot continuations. The new expressions are **call1cc** (which captures a continuation) and **throw** (which invokes a continuation).

$$\begin{aligned} e \;\rightarrow\; & l \mid x \mid \lambda x \cdot e \mid e\, e \mid x := e \mid \textbf{if}\, e\, \textbf{then}\, e\, \textbf{else}\, e \mid e = e \mid \textbf{nil} \mid \\ & \textbf{call1cc}\, e \mid \textbf{throw}\, v \end{aligned}$$

The evaluation context is extended accordingly.

$$\begin{aligned} C \;\rightarrow\; & \Box \mid e\, C \mid C\, v \mid x := C \mid \textbf{if}\, C\, \textbf{then}\, e\, \textbf{else}\, e \mid C = e \mid v = C \mid \\ & \textbf{call1cc}\, C \end{aligned}$$

A **throw** is not intended to be used directly by a programmer. It is created by a **call1cc** always with a value as its first operand, but only executes when it receives a second operand. So, we treat **throw** v as a value.

$$v \;\rightarrow\; l \mid \lambda x \cdot e \mid \textbf{nil} \mid \textbf{throw}\, v$$

If we drop the one-shot restriction, the semantics for first-class continuations is straightforward.

$$\begin{aligned} \langle C[\textbf{callcc}\, v],\, \theta\rangle &\;\Rightarrow\; \langle C[v\, (\textbf{throw}\, \lambda y \cdot C[\,y\,])],\, \theta\rangle \\ \langle C[\textbf{throw}\, v_1\, v_2],\, \theta\rangle &\;\Rightarrow\; \langle v_1\, v_2,\, \theta\rangle \end{aligned}$$

As expected, **callcc** calls its parameter with an argument that, when called, reinstalls the continuation that was active when **callcc** was invoked (C).

One problem with that semantics is that it duplicates the continuation C. After a **callcc**, C appears both as the current continuation and as a captured continuation inside the **throw** expression. This makes it quite difficult to ensure that a continuation is called only once. We can solve this difficulty by storing the continuation in the store.

$$\begin{aligned}\langle C[\mathbf{callcc}\, v],\, \theta\rangle &\Rightarrow \langle(\mathbf{throw}\, l)(v\,(\mathbf{throw}\, l)),\, \theta[l \leftarrow \lambda y \cdot C[y]]\rangle, l \notin \mathrm{dom}(\theta)\\ \langle C[\mathbf{throw}\, l\ v],\, \theta\rangle &\Rightarrow \langle \theta(l)\, v,\, \theta\rangle\end{aligned}$$

It is easy to see that this semantics is equivalent to the previous one. In both semantics, if the continuation is ever invoked, the result will be $(\lambda y \cdot C[y])\, v$. Otherwise, if the original argument to **callcc** returns, the result expression in the second semantics will be again $(\lambda y \cdot C[y])\, v$, which reduces in two steps to the result of the first semantics, $C[v]$.

Now, to ensure one-shotness, we redefine **throw** to invalidate its label after using it.

$$\langle C[\mathbf{call1cc}\, v],\, \theta\rangle \Rightarrow \langle(\mathbf{throw}\, l)(v\,(\mathbf{throw}\, l)),\, \theta[l \leftarrow \lambda y \cdot C[y]]\rangle, l \notin \mathrm{dom}(\theta) \quad (17)$$

$$\langle C[\mathbf{throw}\, l\ v],\, \theta\rangle \Rightarrow \langle \theta(l)\, v,\, \theta[l \leftarrow \mathbf{nil}]\rangle \quad (18)$$

As usual, we say that $e \Downarrow v$ in λ_{c1cc} if $\langle e,\, \theta_0\rangle \overset{*}{\Rightarrow} \langle v,\, \theta\rangle$, for some store θ.

The implementation of one-shot continuations described by Bruggeman et al. [1996] reveals the many similarities between that mechanism and symmetric coroutines. In that 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 5.1. This implementation is shown in Figure 6.

To simulate that semantics on top of λ_{sym} (symmetric coroutines), we use the following definition for $call1cc$.

```
let call1cc = λf.
        let cc = current in
        let throw = λval · (let curr = cc in cc := nil; transfer curr val) in
            transfer (create λc · c (f c)) throw
in · s
```

This code is a straightforward translation of the Lua code at Figure 6.

```
function call1cc(f)
  local cc = current

  local throw = function(val)
    local curr = cc
    cc = nil
    sym.transfer(curr, val)
  end

  return sym.transfer(sym.create(function(c) c(f(c)) end), throw)
end
```

Fig. 6. Implementing one-shot continuations with symmetric coroutines.

Again we may prove the correctness of this definition by mapping states in λ_{sym} to states in λ_a, and showing that the transition rules preserve the mapping. In Ierusalimschy and de Moura [2008] we present this proof.

5.3 One-Shot Subcontinuations

Despite their expressive power, traditional continuations, either multishot or one-shot, are difficult to use; except for some trivial examples, they complicate considerably the structure of programs. Most of the complexity involved in the use of continuations arises from the fact that they represent the *whole* rest of a computation. The convenience of limiting the extent of continuations and localizing the effect of their control operators motivated the introduction of *partial* or *delimited* continuations [Felleisen 1988; Johnson and Duggan 1988] and the proposal of a series of constructs based on this concept [Danvy and Filinski 1990; Queinnec and Serpette 1991; Sitaram 1993; Hieb et al. 1994]. The essence of these abstractions is that the invocation of a captured delimited continuation does not abort the current continuation; instead, delimited continuations can be composed like regular functions.

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

The semantics of subcontinuations can be described with another extension of our core language, which we will call λ_{subc}. This extended language incorporates *labeled expressions* and two control operators: **spawn**, which creates and starts a subcomputation, and **controller**, which invokes a controller.

$$\begin{array}{rcl} e & \rightarrow & l \mid x \mid \lambda x \cdot e \mid e\, e \mid x := e \mid \textbf{if}\, e\, \textbf{then}\, e\, \textbf{else}\, e \mid e = e \mid \textbf{nil} \mid \\ & & l\colon e \mid \textbf{spawn}\, e \mid \textbf{controller}\, l \end{array}$$

Like **throw** in λ_{c1cc}, the operator **controller** is not intended to be used directly by a programmer. As we will see next, it is created by the evaluation of **spawn**, with a specific label as its first argument, which identifies the corresponding subcomputation. It only executes when it receives a second argument which represents the procedure to be applied to the captured subcontinuation. We then treat **controller** l as a value in λ_{subc}.

$$v \rightarrow l \mid \lambda x \cdot e \mid \textbf{nil} \mid \textbf{controller}\, l$$

We use the following definition for the evaluation contexts of λ_{subc}.

$$\begin{array}{rcl} C & \rightarrow & \square \mid e\, C \mid C\, v \mid x := C \mid \textbf{if}\, C\, \textbf{then}\, e\, \textbf{else}\, e \mid C = e \mid v = C \mid \\ & & l\colon C \mid \textbf{spawn}\, C \end{array}$$

The semantics of subcontinuations is described by the rules shown next:[12]

$$\begin{array}{rcll} \langle C[\textbf{spawn}\, v],\ \theta\rangle & \Rightarrow & \langle C[l\colon v\,(\lambda x \cdot \textbf{controller}\, l\colon x)],\ \theta\rangle & (19) \\ & & \text{where } l\colon \text{is a fresh label} & \\ \langle C[l\colon v],\ \theta\rangle & \Rightarrow & \langle C[v],\ \theta\rangle & (20) \\ \langle C_1[l\colon C_2[\textbf{controller}\, l\ v]],\ \theta\rangle & \Rightarrow & \langle C_1[v\,(\lambda x \cdot l\colon C_2[x])],\ \theta\rangle & (21) \end{array}$$

Rule 19 describes the semantics of the operator **spawn**. It installs a new (fresh) label, producing a labeled expression—a subcomputation—which invokes **spawn**'s argument with a controller associated with that label.

Rule 20 shows what happens when a subcomputation ends without invoking the controller: Its label is removed and its result value is returned to its last invocation point.

Rule 21 describes the action of invoking a controller, showing how a subcontinuation is created. A controller invocation must occur within a labeled expression with a matching label (an active subcomputation). The captured subcontinuation is an abstraction created from the context of that subcomputation, including the matching label. The second argument provided to the operator **controller** is applied to that subcontinuation; this application occurs in a context that does not include the abstracted context.

When the restriction imposed to one-shot continuations (a single invocation) is applied to subcontinuations, we have the concept of one-shot subcontinuations [Kumar et al. 1998]. To describe the semantics of one-shot subcontinuations, we can use the same technique that we used to ensure one-shotness for first-class continuations. First we extend λ_{subc} with a new operator (**subcont**) that invokes a subcontinuation. The operator **subcont** is always associated with a specific label, which identifies a subcomputation. The expression

[12]Except for some syntactical adaptations, this is the operational semantics of subcontinuations developed by Hieb et al. [1994].

subcont l is treated as a value.

$$v \;\rightarrow\; \ldots \mid \mathbf{subcont}\, l$$

We then redefine the rewrite rules to ensure that a subcontinuation can be invoked only once.

$$\begin{aligned}
\langle C[\mathbf{spawn}\, v],\, \theta\rangle &\Rightarrow \langle C[l\colon v\,(\lambda x \cdot \mathbf{controller}\, l\colon x)],\, \theta[l \leftarrow \mathbf{nil}]\rangle && (22)\\
&\ \text{where } l \text{ is a fresh label} &&\\
\langle C[l\colon v],\, \theta\rangle &\Rightarrow \langle C[v],\, \theta\rangle && (23)\\
\langle C_1[l\colon C_2[\mathbf{controller}\, l\; v]],\, \theta\rangle &\Rightarrow \langle C_1[v\,(\mathbf{subcont}\, l)],\, \theta[l \leftarrow \lambda x \cdot C_2[x]]\rangle && (24)\\
\langle C[\mathbf{subcont}\, l\; v],\, \theta\rangle &\Rightarrow \langle C[l\colon \theta(l)\, v],\, \theta[l \leftarrow \mathbf{nil}]\rangle && (25)
\end{aligned}$$

Rules 22 and 23 are similar to rules 19 and 20; the only difference is that rule 22 clarifies the notion of a fresh label by stating that a fresh label is not present in the store.

According to rule 24, when a controller is invoked the captured subcontinuation is saved in the store, mapped to the label that represents the corresponding subcomputation. The procedure passed to the controller receives a **subcont** expression that can be used to invoke this subcontinuation.

Rule 25 shows that the invocation of a subcontinuation invalidates the mapping of its corresponding label; this prevents a subcontinuation from being shot more than once.

When we compare the semantics of one-shot subcontinuations with the semantics of full asymmetric coroutines (described in Section 3.2), we can observe many similarities. A full asymmetric coroutine can be seen as an independent subcomputation. Spawning a subcomputation is similar to creating and activating an asymmetric coroutine. Except for the application of the controller argument to the captured subcontinuation, invoking a subcomputation controller (rule 24) is very much like suspending an asymmetric coroutine (rule 10). Invoking a one-shot subcontinuation (rule 24) is also similar to resuming an asymmetric coroutine (rule 9).

The main difference between subcontinuations and other types of delimited continuations is that a 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.[13] This facility makes subcontinuations a useful abstraction for controlling tree-structured concurrency, allowing nonlocal exits to arbitrary points in a process tree: the scenario which motivated the introduction of subcontinuations, formerly called *process continuations* [Hieb and Dybvig 1990].

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

The following definition of *spawn* simulates the semantics of one-shot subcontinuations on top of λ_a.

```
$\mathbf{let}\, spawn = \lambda_f \cdot$
        $\mathbf{let}\, controller = nil \;\; \mathbf{in}$
        $\mathbf{let}\, subcomp = \mathbf{create}\, \lambda c \cdot C_t[f(c)] \;\; \mathbf{in}$
        $\mathbf{let}\, subcont = \lambda x \cdot \mathbf{resume}\, subcomp\; x \;\; \mathbf{in}$
        $\mathbf{let}\, invokecontroller = \lambda_g \cdot ($
          $subcont := (\lambda x \cdot \mathbf{resume}\, subcomp\; x);$
          $controller := nil;$
          $C_y[\mathbf{yield}\, \lambda x \cdot g(x)])\; \mathbf{in}$
        $\mathbf{let}\, reinstate = nil \;\; \mathbf{in}$
          $reinstate := \lambda x \cdot (controller := invokecontroller; C_r[subcont\; x]);$
          $reinstate(\lambda_h \cdot controller\; h)$
$\mathbf{in}\, spawn$
```

Our definition of *spawn* makes use of three auxiliary contexts. The context C_t implements the actions that need to be performed when a subcomputation terminates: the invalidation of its controller and the return of the value produced by the subcomputation to its (re)activation point. Its definition is given next.

$$\mathbf{let}\, x = \Box \;\; \mathbf{in}\, controller := \mathbf{nil}; \lambda y \cdot x$$

The context C_y represents a continuation point of a subcomputation; it is executed when a subcontinuation is invoked, signaling that the subcontinuation has been shot. It is defined as follows.

$$\mathbf{let}\, x = \Box \;\; \mathbf{in}\, subcont := \mathbf{nil};\; x$$

The context C_r is the continuation point of a subcomputation (re)activation; it is executed when the subcomputation ends or invokes a controller. In order to express subcontinuations that are composed by an arbitrary number of nested subcomputations, we need to determine the subcomputation that corresponds to the invoked controller and successively suspend all its nested subcomputations until the controller root is reached. By doing this, we include these nested subcomputations in the captured subcontinuation. When this subcontinuation is invoked we can reinstate its corresponding subcomputation by successively resuming the suspended subcomputations, in the reverse order of their suspension, until the original controller invocation point is reached. To express this behavior, we define the context C_r as in the following.

```
$\mathbf{let}\, x = \Box \;\; \mathbf{in}$
          $\mathbf{if}\, controller\; \mathbf{then}\, reinstate(invokecontroller(x))\; \mathbf{else}\, x(reinstate)$
```

Figure 7 shows a direct translation of the definition of *spawn* in λ_a to Lua code. Again, we may prove the correctness of this definition by mapping states in λ_{sym} to states in λ_a, and showing that the transition rules preserve the mapping. In Ierusalimschy and de Moura [2008] we sketch a proof of correctness for this definition.

The basic idea of our implementation of one-shot subcontinuations using asymmetric coroutines is somewhat similar to the implementation described

```
function spawn(f)
  local controller, subcont, reinstate, invokecontroller

  local subcomp = coroutine.wrap(function(c)
                                   local x = f(c)
                                   controller = nil
                                   return function() return x end
                                 end)
  subcont = function(x) return subcomp(x) end

  function invokecontroller(g)
    subcont = function(x) return subcomp(x) end
    controller = nil
    local x = coroutine.yield(function(x) return g(x) end)
    subcont = nil
    return x
  end

  function reinstate(val)
    controller = invokecontroller
    local ret = subcont(val)
    if controller == nil then
      return ret(reinstate)
    else
      return reinstate(invokecontroller(ret))
    end
  end

  return reinstate(function(h) return controller(h) end)
end
```

Fig. 7. Implementing one-shot subcontinuations with Lua asymmetric coroutines.

by Kumar et al. [1998]. In their work, a subcomputation is represented by a child thread which is created when **spawn** is invoked; synchronization mechanisms based on condition variables and a mutex support the suspension and (re)activation of the subcomputation and its invoker.

6. FINAL REMARKS

After a period of intense investment, from the middle 1960's to the early 1980's, 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 these are not powerful enough to constitute a general control abstraction; in particular, they cannot be used as a concurrent construct.

In this article 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 also 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.

Finally, we demonstrated that full asymmetric coroutines can express symmetric coroutines and vice versa. We also demonstrated that full coroutines can express both one-shot continuations and one-shot delimited continuations; therefore, they can provide any sort of control structure implemented by those constructs. We also discussed the similarities between one-shot continuations and full symmetric coroutines and between one-shot delimited continuations and full asymmetric coroutines. Like delimited continuations, asymmetric coroutines are composable; this similar behavior provides similar benefits: easiness to manage and understand, and support for elegant and concise implementations of several control behaviors.

As with any formal model, it is important to understand the limitations of our model. One limitation is that it does not address the problem of interlanguage calls, wherein one of the languages does not support coroutines (which is the case of programs written in Lua and C, for instance). We could model this situation by defining a new kind of context (to represent pending calls in the foreign language), and a restriction that contexts of this new kind cannot be saved in the store. This model would make clear that any program using symmetric coroutines that have pending calls to the foreign language in its main coroutine cannot do any transfer, because this would involve saving an invalid context in the store. Nevertheless, this restriction does not exist when the same program runs simulated with asymmetric coroutines because the simulation never saves the main continuation (instead, it keeps this continuation in the main expression).

Another point not addressed by our model is performance. In our implementation of one-shot continuations, a single coroutine (i.e., a single stack "segment") is 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 (e.g., the implementation described by Bruggeman et al. [1996]). On the other hand, the implementation of coroutines with multi-shot 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; hence, it may perform less efficiently and use more memory than a direct implementation of coroutines.

It is interesting to compare our notion of expressive power with Felleisen [1990]. His definition allows transformations in the leaves of a program (what he calls *macro expressibility*), while ours allows transformations at the root (therefore allowing *function expressibility*). It is clear that both definitions keep intact the program structure; the enclosing of a program e in a fixed context to bring $C[e]$ clearly does not require "a global reorganization of the entire program." For our purposes Felleisen's definition is not sufficient because we need some form of global state for some simulations (e.g., to keep the current coroutine label when implementing symmetric coroutines).

Our work uses the external context only to introduce global bindings. Hieb et al. [1994] present a different case where our framework can be used. There, to implement full continuations using subcontinuations, they need to enclose the whole program in a call to `spawn`, "to establish the root of the entire computation." This enclosing (plus a suitable definition for `call/cc`) can be expressed using an enclosing context, but cannot be expressed using macros.

One thing we did not explore is negative results. For instance, we did not prove that we cannot express full coroutines using restricted forms of coroutines, such as nonfirst-class or nonstackful coroutines. We also did not prove that we cannot express multishot continuations using coroutines. Intuitively, we can argue for both results based on how these languages handle contexts. For instance, while the rewrite rule for **callcc** duplicates a context, no rule in λ_a ever does so. With such negative results, we could formalize a hierarchy of control constructs, from restricted forms of coroutines to full coroutines (and one-shot continuations) to multishot 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 the USENIX Annual Technical Conference*. USENIX.

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)*.

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 Conference on Programming Language Design and Implementation (PLDI'96)*. ACM, 99–107. *SIGPLAN Not. 31*, 5.

CLOCKSIN, W. AND MELLISH, C. 1981. *Programming in Prolog*. Springer.

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. *Commun. ACM 6,* 7, 396–408.

DAHL, O.-J., DIJKSTRA, E. W., AND HOARE, C. A. R. 1972. Hierarchical program structures. In *Structured Programming*, 2nd ed. Academic Press, London.

DANVY, O. AND FILINSKI, A. 1990. Abstracting control. In *Proceedings of the ACM Conference on LISP and Functional Programming*. ACM, 151–160.

DYBVIG, R. AND HIEB, R. 1989. Engines from continuations. *Comput. Lang. 14,* 2, 109–123.

FELLEISEN, M. 1985. Transliterating Prolog into Scheme. Tech. rep. 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, 180–190.

FELLEISEN, M. 1990. On the expressive power of programming languages. In *Proceedings of the 3rd European Symposium on Programming (ESOP'90)*, 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.

GRISWOLD, R. AND GRISWOLD, M. 1983. *The Icon Programming Language*. Prentice-Hall, 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, 163–173.

HAYNES, C. T. 1987. Logic continuations. *J. Logic Program. 4*, 157–176.

HAYNES, C. T. AND FRIEDMAN, D. 1987a. Abstracting timed preemption with engines. *J. Comput. Lang. 12,* 2, 109–121.

HAYNES, C. T., FRIEDMAN, D., AND WAND, M. 1986. Obtaining coroutines with continuations. *Comput. Lang. 11,* 3/4, 143–153.

HAYNES, C. T. AND FRIEDMAN, D. P. 1987b. Embedding continuations in procedural objects. *ACM Trans. Progam. Lang. Syst. 9,* 4, 582–598.

HIEB, R. AND DYBVIG, R. 1990. Continuations and concurrency. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 128–136.

HIEB, R., DYBVIG, R., AND ANDERSON III, C. W. 1994. Subcontinuations. *Lisp Symbolic Comput. 7,* 1, 83–110.

IERUSALIMSCHY, R. 2003. *Programming in Lua*. Lua.org, Rio de Janeiro, Brazil.

IERUSALIMSCHY, R. AND DE MOURA, A. L. 2008. Some proofs about coroutines. Monografias em Ciẽncia da Computação 04/08, PUC-Rio, Rio de Janeiro, Brazil.

IERUSALIMSCHY, R., FIGUEIREDO, L., AND CELES, W. 1996. Lua—An extensible extension language. *Softw. Pract. & Exper. 26,* 6, 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.

KELSEY, R., CLINGER, W., AND J. REES, E. 1998. Revised^5 report on the algorithmic language Scheme. *Higher-Order Symbolic Comput. 11,* 1, 7–105.

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 Symbolic Comput. 10,* 3, 223–236.

LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. 1977. Abstraction mechanisms in CLU. *Commun. ACM 20,* 8 (Aug.), 564–576.

MARLIN, C. D. 1980. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Lecture Notes in Computer Science, vol. 95, Springer.

MOODY, K. AND RICHARDS, M. 1980. A coroutine mechanism for BCPL. *Softw. Pract. & Exper. 10,* 10, 765–771.

MOURA, A., RODRIGUEZ, N., AND IERUSALIMSCHY, R. 2004. Coroutines in Lua. *J. Universal Comput. Sci. 10,* 7, 910–925.

MURER, S., OMOHUNDRO, S., STOUTAMIRE, D., AND SZYPERSKI, C. 1996. Iteration abstraction in Sather. *ACM Trans. Progam. Lang. Syst. 18,* 1, 1–15.

PAULI, W. AND SOFFA, M. L. 1980. Coroutine behaviour and implementation. *Softw, Pract. & Exper. 10,* 3, 189–204.

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, 174–184.

RICHTER, J. 1997. *Advanced Windows*, 3rd 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`.

SERRANO, M., BOUSSINOT, F., AND SERPETTE, B. 2004. Scheme fair threads. In *Proceedings of the 6th SIGPLAN International Congress on Principles and Practice of Declarative Programming (PPDP)*, 203–214.

SITARAM, D. 1993. Handling control. In *Proceedings of the ACM SIGPLAN'93 Confrence on Programming Language Design and Implementation (PLDI'93)*. ACM, *SIGPLAN Not. 28*, 6.

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

SPRINGER, G. AND FRIEDMAN, D. 1989. *Scheme and the Art of Programming*. MIT Press, Cambridge, MA.

TISMER, C. 2000. Continuations and stackless Python. In *Proceedings of the 8th International Python Conference*. Arlington, VA. `http://www.python.org/workshops/2000-01/proceedings.html`.

WAND, M. 1980. Continuation-Based multiprocessing. In *Proceedings of the Lisp Conference*. ACM, 19–28. Reprinted in *Higher-Order Symbolic Comput. 12*, 3, 285–299, 1999.

WAND, M. AND VAILLANCOURT, D. 2004. Relating models of backtracking. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*. ACM, 54–65.

WIRTH, N. 1985. *Programming in Modula-2*, 3rd, corrected ed. Springer.

Received March 2008; accepted April 2008