PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 04/08

Some Proofs about Coroutines

Roberto Ierusalimschy
Ana Lúcia de Moura

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 04/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena February, 2008

Some Proofs about Coroutines

Roberto Ierusalimschy and Ana Lúcia de Moura

roberto@inf.puc-rio.br
ana@rnp.br

Abstract:
This paper presents some formal proofs regarding the equivalence of expressive power of asymmetric coroutines, symmetric coroutines, one-shot continuations, and one-shot delimited continuations.

Keywords: Coroutines, operational semantics, continuations

Resumo:
Este artigo apresenta algumas provas relativas a equivalência do poder expressivo de co-rotinas assimétricas, co-rotinas simétricas, continuações *one shot* e continuações delimitadas *one shot*.

Palavras-chave: Co-rotinas, semântica operacional, continuações

In charge for publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ, Brazil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

In a related paper [dMI09], we advocated the revival of coroutines as a simple yet powerful control mechanism. In that paper, we claimed that coroutines are at least as expressive as one-shot continuations, and also that two different forms of coroutines, symmetric and asymmetric, have equivalent expressive power. In this paper, we present some formal proofs sustaining those claims. To make this paper self contained, we borrowed from [dMI09] some explanations and definitions.

The remainder of this paper is organized as follows. Section 2 provides a formal description of our concept of full asymmetric coroutines. In section 3 we demonstrate that full asymmetric coroutines can provide not only symmetric coroutine facilities but also one-shot continuations and one-shot delimited continuations. Section 4 summarizes the paper and presents some final remarks.

2 Full Asymmetric Coroutines

This section presents an operational semantics for a simple language that incorporates asymmetric coroutines.

2.1 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 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, 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.

2.2 Operational Semantics

We present here an operational semantics for the mechanisms we just described. Our approach is similar to the operational semantics of *subcontinuations*, described in [HDA94]. 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:

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

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

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

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

When needed, we will use the usual syntactic sugar $\textbf{let}\, x = e_1 \,\textbf{in}\, e_2$ meaning $(\lambda x.\, e_2)\, e_1$ (where x may occur free in e_2); and $e_1; e_2$ meaning $(\lambda x.\, e_2)\, e_1$ for some x not free in e_2. The precedence of the operators is as follows, from higher to lower:

- application
- assignment
- conditionals
- semicolon
- lambda abstractions
- let expressions

A store θ, mapping variables and labels to values, is included in the definition of the core language to allow side-effects:[1]

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

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

$$\begin{aligned} FV(\theta) &= \bigcup_{x \in \mathrm{dom}(\theta)} FV(\theta(x)) \\ LB(\theta) &= \bigcup_{x \in \mathrm{dom}(\theta)} LB(\theta(x)) \end{aligned}$$

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. We use *evaluation contexts* [FF86] 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[2] of applications:

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

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

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

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

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

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 guarantees the use of a fresh variable z. Rule 3, which describes the semantics of assignments, assumes that the variable already exists in the store (i.e., it was previously introduced by an abstraction). Rule 4 and 5 describe conditionals, which test whether the condition is **nil** or not 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 $\Rightarrow$.

It is easy to check that our language, up to now, is deterministic: given an expression e, there is at most one expression e_1 such that $e \Rightarrow e_1$. (In particular, it has a unique decomposition property.) However, to simplify some proofs presented later, we will introduce a garbage-collection rule [MFH95] that breaks this determinism:

$$\langle e,\, \theta \cup \theta'\rangle \;\;\Rightarrow\;\; \langle e,\, \theta\rangle,\ \mathrm{dom}(\theta') \cap (FV(\theta) \cup FV(e) \cup LB(\theta) \cup LB(e)) = \{\,\} \qquad (8)$$

This rule simply allows us to remove from the store any set of bindings that are not referred to by other expressions in a state. Again it is easy to check that this rule does not break the determinism of results; given any expression e, there is still at most one value v such that $e \Downarrow v$.

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

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

The precedence of the label is lower than that of the semicolon, so $l\colon a; b$ reads as $l\colon (a; b)$ instead of $(l\colon a); b$. 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 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 need 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).

Next we give the rewrite rules that describe the semantics of the coroutine operators:

$$\begin{aligned}
\langle C[\textbf{create}\ v],\, \theta\rangle &\;\;\Rightarrow\;\; \langle C[l],\, \theta[l \leftarrow v]\rangle,\ l \notin (LB(\theta) \cup LB(C[\textbf{create}\ v])) && (9)\\
\langle C[\textbf{resume}\ l\ v],\, \theta\rangle &\;\;\Rightarrow\;\; \langle C[l\colon \theta(l)\ v],\, \theta[l \leftarrow \textbf{nil}]\rangle && (10)\\
\langle C_1[l\colon C_2'[\textbf{yield}\ v]],\, \theta\rangle &\;\;\Rightarrow\;\; \langle C_1[v],\, \theta[l \leftarrow \lambda x.\, C_2'[x]]\rangle && (11)\\
\langle C[l\colon v],\, \theta\rangle &\;\;\Rightarrow\;\; \langle C[v],\, \theta\rangle && (12)
\end{aligned}$$

Rule 9 describes the action of creating a coroutine. It creates a fresh label to represent the coroutine and stores a mapping from this label to the coroutine body.

Rule 10 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** in the resulting store.

Rule 11 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.

3 Expressing Alternative Control Structures

In this section we 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 $\{a_1, \cdots, a_n\}$ and the other a set $\{b_1, \cdots, b_m\}$, 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.

3.1 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 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.[3]

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 now formalize our model of symmetric coroutines. We do that by extending the core language introduced in Section 2.2. We will call this extended language λ_{sym}. We begin by extending the set of expressions with the symmetric coroutine operators:

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

[3]This is a somewhat arbitrary behaviour. A program could as well terminate upon the termination of any of its coroutines.

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

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

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 1–8 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:

$$\langle C[\textbf{create}\ v],\,\theta,\,l_1\rangle \;\Rightarrow\; \langle C[l_2],\,\theta[l_2 \leftarrow v],\,l_1\rangle,\ \text{where } l_2 \text{ is a fresh label} \tag{13}$$

$$\langle C[\textbf{transfer}\ l_2\ v],\,\theta,\,l_1\rangle \;\Rightarrow\; \langle \theta(l_2)\ v,\,\theta[l_2 \leftarrow \textbf{nil},\ l_1 \leftarrow \lambda x.\,C[x]],\,l_2\rangle,\ l_1 \neq l_2 \tag{14}$$

$$\langle C[\textbf{transfer}\ l\ v],\,\theta,\,l\rangle \;\Rightarrow\; \langle C[v],\,\theta,\,l\rangle \tag{15}$$

$$\langle C[\textbf{current}],\,\theta,\,l\rangle \;\Rightarrow\; \langle C[l],\,\theta,\,l\rangle \tag{16}$$

$$\langle v,\,\theta,\,l\rangle \;\Rightarrow\; \langle \theta(l_I)\ v,\,\theta[l_I \leftarrow \textbf{nil}],\,l_I\rangle,\ l \neq l_I \tag{17}$$

Rule 13 describes the semantics of creating a symmetric coroutine; this operation is similar to the creation of an asymmetric coroutine, described in Section 2.2.

Rule 14 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 15 handles the particular case where a coroutine transfers control to itself; this operation simply returns the given value.

Rule 16 provides the semantics for the **current** primitive.

Rule 17 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. The translation of these ideas into the language λ_a gives us the following definition:[4]

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

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

where the context C_d is defined as

let $val = \square$ **in** $current{:}= next; disp\ val$

Our goal now is to prove that, with this definition for *current* and *transfer*, any expression of λ_{sym} keeps its meaning in λ_a. More precisely, we want to prove that, whenever e results in v in λ_{sym}, then **let** $current = \cdots$ **in** e also results in v in λ_a.

For such proof, we will define a mapping Γ from states in λ_{sym} to states in λ_a. This mapping is only defined for reachable states. A state $\langle e, \theta, l\rangle$ is *reachable* if $\langle e_1, \theta_0, l_I\rangle \overset{*}{\Rightarrow} \langle e, \theta, l\rangle$. Reacheable states satisfy the following property:

Lemma 1 *In any reachable state* $\langle e, \theta, l\rangle$*,* $\theta(l) = \mathbf{nil} \wedge (l = l_I \vee \theta(l_I) \neq \mathbf{nil})$.

Proof: The initial label l_I is not defined in the initial empty state θ_0, so the property is trivially true in the initial state. It is easy to check that all rules preserve the property. $\square$

The mapping Γ needs to add some elements to the store of a state. For any store θ, we define a store $\overline{\theta}$ as follows:

$$\overline{\theta} = \theta[main \leftarrow l_I, next \leftarrow l_I, disp \leftarrow (\lambda val.\ \cdots), transfer \leftarrow (\lambda co.\,\lambda val.\ \cdots)]$$

In words, $\overline{\theta}$ is the store θ augmented with the definitions created by *transfer*. The names *main*, *next*, and *disp* are local to the definition of *transfer*, and so can be α-renamed to avoid conflicts, if needed. We will assume that $transfer, current \notin \mathrm{dom}(\theta)$, as both *transfer* and *current* are kind of "reserved words" in λ_{sym}. (This assumption is easily enforced by α-renamings of conflicting variables.)

The mapping Γ for reachable states is defined as follows:

$\Gamma\langle e, \theta[l_I \leftarrow \mathbf{nil}], l_I\rangle = \langle e, \overline{\theta}[current \leftarrow l_I]\rangle$
$\Gamma\langle e, \theta[l_I \leftarrow \lambda x.\, C[x]], l\rangle = \langle C[C_d[l{:}\, e]], \overline{\theta}[current \leftarrow l]\rangle, l \neq l_I$

The first case handles states where the main coroutine is active; the second case handles other states. A detail of the mapping Γ is the treatment of the main coroutine. While the symmetric coroutines semantics saves the main continuation C in the store, associated to l_I, the asymmetric coroutines semantics keeps it in the main expression, because it has no way to capture this continuation when simulating a *transfer*. The sub-continuation C_d (the dispatcher) decides whether the computation should return to that main continuation or resume another coroutine.

Lemma 2 *If* $\langle e_1, \theta_1, l_1\rangle \overset{1}{\Rightarrow} \langle e_2, \theta_2, l_2\rangle$ *in* λ_{sym} *then* $\Gamma\langle e_1, \theta_1, l_1\rangle \overset{*}{\Rightarrow} \Gamma\langle e_2, \theta_2, l_2\rangle$ *in* λ_a.

Proof: It is easy to see that, whenever the main expression in λ_{sym} has the form $C_1[e]$, the main expression in λ_a either has the same form or has the form $C_2[C_d[l{:}\, C_1[e]]]$; in any case e is the next subexpression to be evaluated.[5]

The only rules from λ_{sym} not present in λ_a are Rules 14–15 (transfer), 16 (**current**), and 17 (coroutine termination).

Rule 16 is trivial, because the mapping Γ ensures that *current* always contains the current coroutine.

[5]Here is where we need right-to-left evaluation. In λ_{sym}, (**transfer** e_1 e_2) is a primitive, and its rule (14) only fires when both expressions are already reduced to values. When we translate that to λ_a, it becomes a regular function application, ((*transfer* e_1) $e2$). With left-to-right evaluation, (*transfer* e_1) would be evaluated before e_2, and so the reduction steps in the simulation would not mimic the original steps. With right-to-left evaluation we ensure that the simulation, like the original semantics, will first evaluate the second expression, then the first expression, and only then will apply *transfer*.

For Rule 14 we must consider three cases: transfers from the main coroutine, transfers to the main coroutine, and transfers not involving the main coroutine.

For transfers from the main coroutine we have the following sequence of steps in λ_a:

$\Gamma\langle C[\mathbf{transfer}\ l\ v],\ \theta,\ l_I\rangle =$
$\langle C[\mathit{transfer}\ l\ v],\ \overline{\theta}[\mathit{current} \leftarrow l_I]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathit{current}{:=}\ l;\ \mathit{disp}\ \mathit{val}],\ \overline{\theta}[\mathit{current} \leftarrow l_I, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathit{disp}\ v],\ \overline{\theta}[\mathit{current} \leftarrow l]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathbf{if}\ \mathit{current} = \mathit{main}\ \mathbf{then}\ \mathit{val}\ \mathbf{else}\ (\mathit{next}{:=}\ \mathit{main};\ C_d[\mathbf{resume}\ \mathit{current}\ \mathit{val}])],$
$\qquad \overline{\theta}[\mathit{current} \leftarrow l, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C[C_d[\mathbf{resume}\ l\ v]],\ \overline{\theta}[\mathit{current} \leftarrow l]\rangle \overset{*}{\Rightarrow}$
$\langle C[C_d[l{:}\,\overline{\theta}(l)\ v]],\ \overline{\theta}[\mathit{current} \leftarrow l, l \leftarrow \mathbf{nil}]\rangle =$
$\Gamma\langle \theta(l)\ v,\ \theta[l \leftarrow \mathbf{nil}, l_I \leftarrow \lambda x.\ C[x]],\ l\rangle$

It is easy to see that this corresponds to Rule 14. Note that the last sequence of steps may include several uses of Rule 8 to clear the store from entries created by applications (e.g., *val*).

For transfers to the main coroutine we have the following reductions:

$\Gamma\langle C_1[\mathbf{transfer}\ l_I\ v],\ \theta[l_I \leftarrow \lambda x.\ C_2[x]],\ l\rangle =$
$\langle C_2[C_d[l{:}\ C_1[\mathit{transfer}\ l_I\ v]]],\ \overline{\theta}[\mathit{current} \leftarrow l]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[C_d[l{:}\ C_1[\mathit{next}{:=}\ l_I; \mathbf{yield}\ \mathit{val}]]],\ \overline{\theta}[\mathit{current} \leftarrow l, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[C_d[v]],\ \overline{\theta}[\mathit{current} \leftarrow l, l \leftarrow \lambda x.\ C_1[x]]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[\mathit{current}{:=}\ \mathit{next};\ \mathit{disp}\ \mathit{val}],\ \overline{\theta}[\mathit{current} \leftarrow l, l \leftarrow \lambda x.\ C_1[x], \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[\mathbf{if}\ \mathit{current} = \mathit{main}\ \mathbf{then}\ \mathit{val}\ \mathbf{else}\ \cdots],$
$\qquad \overline{\theta}[\mathit{current} \leftarrow l_I, l \leftarrow \lambda x.\ C_1[x], \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[v],\ \overline{\theta}[\mathit{current} \leftarrow l_I, l \leftarrow \lambda x.\ C_1[x]]\rangle =$
$\Gamma\langle C_2[v],\ \theta[l_I \leftarrow \mathbf{nil}, l \leftarrow \lambda x.\ C_1[x]],\ l_I\rangle$

For other transfers, our simulation in λ_a does the following:

$\Gamma\langle C_1[\mathbf{transfer}\ l_2\ v],\ \theta[l_I \leftarrow \lambda x.\ C_2[x]],\ l_1\rangle =$
$\langle C_2[C_d[l_1{:}\ C_1[\mathit{transfer}\ l_2\ v]]],\ \overline{\theta}[\mathit{current} \leftarrow l_1]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[C_d[l_1{:}\ C_1[\mathit{next}{:=}\ l_2; \mathbf{yield}\ \mathit{val}]]],\ \overline{\theta}[\mathit{current} \leftarrow l_1, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[C_d[l_1{:}\ C_1[\mathbf{yield}\ v]]],\ \overline{\theta}[\mathit{current} \leftarrow l_1, \mathit{next} \leftarrow l_2]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[C_d[v]],\ \overline{\theta}[\mathit{current} \leftarrow l_1, \mathit{next} \leftarrow l_2, l_1 \leftarrow \lambda x.\ C_1[x]]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[\mathit{current}{:=}\ \mathit{next};\ \mathit{disp}\ \mathit{val}],$
$\qquad \overline{\theta}[\mathit{current} \leftarrow l_1, \mathit{next} \leftarrow l_2, l_1 \leftarrow \lambda x.\ C_1[x], \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[\mathbf{if}\ \mathit{current} = \mathit{main}\ \mathbf{then}\ \mathit{val}\ \mathbf{else}\ (\mathit{next}{:=}\ \mathit{main};\ C_d[\mathbf{resume}\ \mathit{current}\ \mathit{val}])],$
$\qquad \overline{\theta}[\mathit{current} \leftarrow l_2, \mathit{next} \leftarrow l_2, l_1 \leftarrow \lambda x.\ C_1[x], \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[C_d[\mathbf{resume}\ l_2\ \mathit{val}]],\ \overline{\theta}[\mathit{current} \leftarrow l_2, l_1 \leftarrow \lambda x.\ C_1[x], \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C_2[C_d[l_2{:}\,\overline{\theta}(l_2)\ v]],\ \overline{\theta}[\mathit{current} \leftarrow l_2, l_1 \leftarrow \lambda x.\ C_1[x], l_2 \leftarrow \mathbf{nil}]\rangle =$
$\Gamma\langle \theta(l_2)\ v,\ \theta[l_I \leftarrow \lambda x.\ C_2[x], l_1 \leftarrow \lambda x.\ C_1[x], l_2 \leftarrow \mathbf{nil}],\ l_2\rangle$

Again this corresponds to Rule 14.

For Rule 15 we must consider two cases: transfers from the main coroutine to itself and transfers from a non-main coroutine to itself. For transfers from the main coroutine to itself we have the following sequence of steps in λ_a:

$\Gamma\langle C[\mathbf{transfer}\ l_I\ v],\ \theta,\ l_I\rangle =$
$\langle C[\mathit{transfer}\ l_I\ v],\ \overline{\theta}[\mathit{current} \leftarrow l_I]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathit{current}{:=}\ l_I;\ \mathit{disp}\ \mathit{val}],\ \overline{\theta}[\mathit{current} \leftarrow l_I, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathit{disp}\ v],\ \overline{\theta}[\mathit{current} \leftarrow l_I]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathbf{if}\ \mathit{current} = \mathit{main}\ \mathbf{then}\ \mathit{val}\ \mathbf{else}\ \cdots],\ \overline{\theta}[\mathit{current} \leftarrow l_I, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C[v],\ \overline{\theta}[\mathit{current} \leftarrow l_I]\rangle =$
$\Gamma\langle C[v],\ \theta,\ l_I\rangle$

Transfers from non-main coroutines to themselves follow the same sequence of steps of Rule 14.

Finally, for Rule 17, when a coroutine terminates (that is, evaluates to a value), our simulation in λ_a does the following:

$\Gamma\langle v,\, \theta[l_I \leftarrow \lambda x.\, C[x]],\, l\rangle =$
$\langle C[C_d[l\colon v]],\, \overline{\theta}[\mathit{current} \leftarrow l]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathit{current}\colon\!= \mathit{next};\, \mathit{disp}\ \mathit{val}],\, \overline{\theta}[\mathit{current} \leftarrow l, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C[\mathbf{if}\ \mathit{current} = \mathit{main}\ \mathbf{then}\ \mathit{val}\ \mathbf{else}\ \cdots],\, \overline{\theta}[\mathit{current} \leftarrow l_I, \mathit{val} \leftarrow v]\rangle \overset{*}{\Rightarrow}$
$\langle C[v],\, \overline{\theta}[\mathit{current} \leftarrow l_I]\rangle =$
$\Gamma\langle C[v],\, \theta[l_I \leftarrow \mathbf{nil}],\, l_I\rangle$

□

Lemma 3 *If $\langle e_1,\, \theta_1,\, l_1\rangle \overset{*}{\Rightarrow} \langle e_2,\, \theta_2,\, l_2\rangle$ in λ_{sym} then $\Gamma\langle e_1,\, \theta_1,\, l_1\rangle \overset{*}{\Rightarrow} \Gamma\langle e_2,\, \theta_2,\, l_2\rangle$ in λ_a.*

Proof: Induction on the number of steps and Lemma 2.

□

Theorem 1 *If $e \Downarrow v$ in λ_{sym}, then ($\mathbf{let}$ $\mathit{transfer} = \cdots$ $\mathbf{in}$ e) $\Downarrow v$ in λ_a. Moreover, if e diverges in λ_{sym}, then $\mathbf{let}$ $\mathit{transfer} = \cdots$ $\mathbf{in}$ e diverges in λ_a.*

Proof: If we follow the first steps of the evaluation of **let** *current* $= \cdots$ **in** e, which evaluate the outer let expressions, we have

$$\langle \mathbf{let}\ \mathit{current} = \cdots\ \mathbf{in}\ e,\, \theta_0\rangle \overset{*}{\Rightarrow} \langle e,\, \overline{\theta}_0[\mathit{current} \leftarrow l_I]\rangle$$

By definition, if $e \Downarrow v$ in λ_{sym}, then $\langle e,\, \theta_0,\, l_I\rangle \overset{*}{\Rightarrow} \langle v,\, \theta,\, l_I\rangle$. So, by Lemma 3, $\Gamma\langle e,\, \theta_0,\, l_I\rangle \overset{*}{\Rightarrow} \Gamma\langle v,\, \theta,\, l_I\rangle$ in λ_a. Now note that $\Gamma\langle e,\, \theta_0,\, l_I\rangle = \langle e,\, \overline{\theta}_0[\mathit{current} \leftarrow l_I]\rangle$ and that $\Gamma\langle v,\, \theta,\, l_I\rangle = \langle v,\, \overline{\theta}[\mathit{current} \leftarrow l_I]\rangle$. Therefore,

$$\langle \mathbf{let}\ \mathit{current} = \cdots\ \mathbf{in}\ e,\, \theta_0\rangle \overset{*}{\Rightarrow} \langle e,\, \overline{\theta}_0[\mathit{current} \leftarrow l_I]\rangle \overset{*}{\Rightarrow} \langle v,\, \overline{\theta}[\mathit{current} \leftarrow l_I]\rangle$$

Now assume an expression e that diverges in λ_{sym}. That means that, given any natural n, there is a derivation $\langle e,\, \theta_0,\, l_I\rangle \overset{m}{\Rightarrow} \langle e_1,\, \theta_1,\, l_1\rangle$, where $m \geq n$. But that means that

$$\Gamma\langle e,\, \theta_0,\, l_I\rangle = \langle e,\, \overline{\theta}_0[\mathit{current} \leftarrow l_I]\rangle \overset{k}{\Rightarrow} \langle e_1',\, \overline{\theta}_1[\mathit{current} \leftarrow l_1]\rangle$$

Because each simulated step involves at least one step in λ_a, we have that $k \geq m \geq n$. Therefore, the derivation length of the simulation is also unbound, meaning that the simulation diverges too.

□

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 this:

```
let $yield$ = nil in
let $resume = \lambda co.\, \lambda val.$
        let $previous$ = current in
        let $oldyield = yield$ in
          $yield\colon\!= \lambda val.\,(yield\colon\!= oldyield;$ 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 that 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 $resume = \lambda co.\, \lambda val.$
        let $previous = $ current in
        let $oldyield = yield$ in
        let $test = nil$ in
          $yield := \lambda val.\, (yield := oldyield; test := \lambda x.\, x;$ transfer $previous\ val);$
          $test := \lambda val.\, test\ (yield\ val);$
          $test$ (transfer $co\ val$)
in $\cdots$
```

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 (that is, its main function returns), *test* yields the returned value (and repeats the test when control eventually returns).

We leave to the reader the proof of correctness of these definitions.

3.2 One-shot Continuations

One-shot continuations [HF87, BWD96] 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`).

To give a formal semantics for one-shot continuations, again we extend our core language to create the language λ_{c1cc}. The new expressions are **call1cc** (which captures a continuation) and **throw** (which invokes a continuation):

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

The evaluation context is extended accordingly:

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

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:

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

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

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

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. That makes it quite difficult to ensure that a continuation is called only once. We can solve that difficulty by storing the continuation in the store:

$$\begin{array}{rcl} \langle C[\textbf{callcc}\ v],\, \theta\rangle & \Rightarrow & \langle(\textbf{throw}\ l)(v\ (\textbf{throw}\ l)),\, \theta[l \leftarrow \lambda y.\ C[y]]\rangle, l \notin \mathrm{dom}(\theta) \\ \langle C[\textbf{throw}\ l\ v],\, \theta\rangle & \Rightarrow & \langle\theta(l)\ v,\, \theta\rangle \end{array}$$

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 $(\lambda y.\ C[y])\ v$. Otherwise, if the original argument to **callcc** returns, the result expression in the second semantics will be again $(\lambda y.\ C[y])\ v$, 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:

$$\begin{array}{rcll} \langle C[\textbf{call1cc}\ v],\, \theta\rangle & \Rightarrow & \langle(\textbf{throw}\ l)(v\ (\textbf{throw}\ l)),\, \theta[l \leftarrow \lambda y.\ C[y]]\rangle, l \notin \mathrm{dom}(\theta) & (18) \\ \langle C[\textbf{throw}\ l\ v],\, \theta\rangle & \Rightarrow & \langle\theta(l)\ v,\, \theta[l \leftarrow \textbf{nil}]\rangle & (19) \end{array}$$

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.

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

```
let $call1cc = \lambda f.$
        let $cc$ = current in
        let $throw = \lambda val.$ (let $curr = cc$ in $cc$:= nil; transfer $curr\ val$) in
            transfer (create $\lambda c.\ c\ (f\ c)$) $throw$
in $\cdots$
```

We will call a λ_{sym} program that uses **create**, **transfer**, and **current** only through calls to *call1cc* a *one-shot simulation.* During a one-shot simulation, each call to *call1cc* creates a new instance of *cc* and a new instance of *throw*. We will use the term cc_i to denote any of these α-renamed instances of *cc*, and $throw_i$ for instances of *throw*. One-shot simulations keep the following important invariant:

Lemma 4 *During the execution of a one-shot simulation, labels are stored only in cc_i variables. Moreover, any label is stored in at most one (live) variable and no stored label refers to the current coroutine.*

Proof: Only *call1cc* and *throw* can access the cc_i variables. Only *call1cc* can call **create** to create new labels. A new label immediately becomes the current coroutine. Calls to *call1cc* assign the value of the current coroutine to a cc_i. So, labels never leak to other variables.

For the uniqueness of variables referring to a given label, we notice that, when a program starts, there are no cc_i variables, so the invariant is vacuously true. When *call1cc* is called, the current label is stored in a cc_i variable, but immediately control goes to a freshly created coroutine, restoring the invariant that no stored label refers to the current coroutine. When *throw* is called, a temporary *curr* variable receives the value of a specific cc_i and cc_i receives **nil**; then *curr* becomes the current coroutine and goes out of scope, restoring the invariant.

□

Now we need a way to relate λ_{c1cc} states to λ_{sym} states. For this end we define a relation

$$\langle e,\, \theta\rangle \cong \langle e,\, \overline{\theta},\, l\rangle$$

between λ_{c1cc} states and λ_{sym} states, where $\overline{\theta}$ is defined as follows:[6]

- $\overline{\theta}$ has all bindings from θ except those indexed by labels;
- $\overline{\theta}$ contains an extra variable *call1cc*, with the function definition, and multiple extra variables $throw_i$ and cc_i, as described next;
- For each sub-expression **throw** l_i in the λ_{c1cc} program, $\overline{\theta}$ contains variables $throw_i$ and cc_i plus a label $\overline{l_i}$ such that, if $\theta(l_i) \neq \textbf{nil}$, then

$$\begin{array}{rcl} \overline{\theta}(throw_i) & = & \lambda val.\,(\textbf{let}\ curr = cc_i\ \textbf{in}\ cc_i{:=}\,\textbf{nil}; \textbf{transfer}\ curr\ val) \\ \overline{\theta}(cc_i) & = & \overline{l_i} \\ \overline{\theta}(\overline{l_i}) & = & \theta(l_i) \end{array}$$

otherwise, when $\theta(l_i) = \textbf{nil}$,

$$\begin{array}{rcl} \overline{\theta}(throw_i) & = & \lambda val.\,(\textbf{let}\ curr = cc_i\ \textbf{in}\ cc_i{:=}\,\textbf{nil}; \textbf{transfer}\ curr\ val) \\ \overline{\theta}(cc_i) & = & \textbf{nil} \end{array}$$

(In this case the value of $\overline{\theta}(\overline{l_i})$ is irrelevant, because there are no references left to $\overline{l_i}$.)

Lemma 4 ensures that each label l_i can be associated to at most one corresponding $throw_i$, so each **throw** l_i translates to a unique set of $throw_i$, cc_i, and l_i. Any change in a given cc_i or l_i cannot affect other parts of the program.

Lemma 5 *If* $\langle e_1, \theta_1\rangle \overset{1}{\Rightarrow} \langle e_2, \theta_2\rangle$ *in* λ_{c1cc} *and* $\langle e_1, \theta_1\rangle \cong \langle e_1, \overline{\theta}_1, l_1\rangle$*, then there is a label* l_2 *such that* $\langle e_1, \overline{\theta}_1, l_1\rangle \overset{*}{\Rightarrow} \langle e_2, \overline{\theta}_2, l_2\rangle$ *in* λ_{sym}.

Proof:

The extra entries of $\overline{\theta}_1$ cannot appear in e_1, so they do not affect transitions other than those made by Rules 18 or 19. Similarly, only those rules involve labels. So, if the transition in λ_{c1cc} is not defined by Rule 18 or 19, λ_{sym} can make the same transition.

For Rule 18, λ_{sym} behaves as follows:

$$\begin{array}{l} \langle C[\textbf{call1cc}\ e],\, \theta\rangle \cong \\ \langle C[call1cc\ e],\, \overline{\theta},\, l_1\rangle \overset{*}{\Rightarrow} \\ \langle C[\textbf{transfer}\ (\textbf{create}\ \lambda c.\ c\ (f\ c))\ throw_i],\, \overline{\theta}[throw_i \leftarrow \cdots, cc_i \leftarrow l_1, f \leftarrow e],\, l_1\rangle \overset{*}{\Rightarrow} \\ \langle C[\textbf{transfer}\ l_i\ throw_i],\, \overline{\theta}[throw_i \leftarrow \cdots, cc_i \leftarrow l_1, f \leftarrow e, l_i \leftarrow \lambda c.\ c\ (f\ c)],\, l_1\rangle \overset{*}{\Rightarrow} \\ \langle throw_i\ (e\ throw_i),\, \overline{\theta}[throw_i \leftarrow \cdots, cc_i \leftarrow l_1, l_1 \leftarrow \lambda x.\ C[x]],\, l_i\rangle \cong \\ \langle(\textbf{throw}\ l_i)(e\ (\textbf{throw}\ l_i)),\, \theta[l_i \leftarrow \lambda x.\ C[x]]\rangle \end{array}$$

For Rule 19, λ_{sym} behaves as follows:

$$\begin{array}{l} \langle C[\textbf{throw}\ l_i\ v],\, \theta[l_i \leftarrow \lambda x.\ C_i[x]]\rangle \cong \\ \langle C[throw_i\ v],\, \overline{\theta}[throw_i \leftarrow \lambda val.\,(\cdots), cc_i \leftarrow l_i, l_i \leftarrow \lambda x.\ C_i[x]],\, l_1\rangle \overset{*}{\Rightarrow} \\ \langle C[cc_i{:=}\,\textbf{nil}; \textbf{transfer}\ curr\ val], \\ \qquad \overline{\theta}[curr \leftarrow l_i, throw_i \leftarrow \lambda v.\,(\cdots), cc_i \leftarrow l_i, l_i \leftarrow \lambda x.\ C_i[x], val \leftarrow v],\, l_1\rangle \overset{*}{\Rightarrow} \\ \langle C[\textbf{transfer}\ l_i\ v],\, \overline{\theta}[throw_i \leftarrow \lambda val.\,(\cdots), cc_i \leftarrow \textbf{nil}, l_i \leftarrow \lambda x.\ C_i[x]],\, l_1\rangle \overset{*}{\Rightarrow} \\ \langle(\lambda x.\ C_i[x])\ v,\, \overline{\theta}[throw_i \leftarrow \lambda val.\,(\cdots), cc_i \leftarrow \textbf{nil}, l_i \leftarrow \textbf{nil}],\, l_i\rangle \cong \\ \langle\theta(l_i)\ v,\, \theta[l_i \leftarrow \textbf{nil}]\rangle \end{array}$$

[6]The main reason this is a relation and not a mapping is that the current coroutine label l in λ_{sym} has no direct relationship with a corresponding state in λ_{c1cc}. Given a state in λ_{c1cc} there is no way to tell which should be the "correct" l in λ_{sym}.

□

With that lemma, we can prove that, whenever a λ_{c1cc} program reaches a final value, its simulation in λ_{sym} also reaches the same value. However, the λ_{sym} program might not be in the main coroutine (label l_I), so it would continue executing the main coroutine. The following lemma proves that this cannot happen.

Lemma 6 *During the execution of a one-shot simulation, at any state $\langle e,\, \theta,\, l\rangle$, either l is the initial label l_I or e has the form $throw_i\; e'$. Moreover, for any $l \in \mathrm{dom}(\theta)$ that has a cc_i pointing to it, either l is the initial label l_I or $\theta(l)$ has the form $\lambda y.\, throw_i\; e'$.*

Proof: When a simulation begins, its label is l_I and there are no labels in the store, so the invariant is true. When a continuation is captured (invocation of *call1cc*), we have basically the following transition:

$$\langle C[call1cc\; e],\, \theta,\, l_1\rangle \overset{*}{\Rightarrow} \langle throw_i\; (e\; (throw_i)),\, \theta[l_1 \leftarrow \lambda x.\, C[x]],\, l_1\rangle$$

The final expression obviously has the form $throw_i\, e'$. For the captured continuation, either $l_1 = l_I$ or $C[x]$ had the form $throw_i\; e'$; in the second case, $\theta(l_1)$ has the form $\lambda y.\, throw_i\; e'$.

When a continuation is invoked, the transition is basically this:

$$\langle C[throw_i\; v],\, \theta[cc_i \leftarrow l_i, l_i \leftarrow \lambda x.\, C_i[x]],\, l_1\rangle \overset{*}{\Rightarrow} \langle C_i[v],\, \theta[cc_i \leftarrow \mathbf{nil}],\, l_i\rangle$$

By the invariant, either $l_i = l_I$ or $C_i[x]$ had the form $throw_i\, e'$. In either case, the invariant is true in the resulting expression. In the resulting state, both labels l_1 and l_i have no cc_i pointing to them (by Lemma 4), so they do not break the invariant.

□

An expression of the form $throw_i\; e$ is not a value in λ_{sym}, so a corollary of the previous lemma is that, in a state $\langle e,\, \theta,\, l\rangle$, if e is a value then $l = l_I$.

Lemma 7 *If $\langle e,\, \theta_0\rangle \overset{*}{\Rightarrow} \langle v,\, \theta_1\rangle$ in λ_{c1cc}, where e is a user-written program, then in λ_{sym} we have that*

$$\langle e,\, \theta_0[call1cc \leftarrow \ldots],\, l_I\rangle \overset{*}{\Rightarrow} \langle v,\, \theta_2,\, l_I\rangle$$

Proof: A user-written program cannot contain **throw** expressions, so we have $\overline{\theta}_0 = \theta_0[call1cc \leftarrow \ldots]$. By induction and Lemma 5, if $\langle e,\, \theta_0\rangle \overset{*}{\Rightarrow} \langle v,\, \theta_1\rangle$, then $\langle e,\, \overline{\theta}_0,\, l_I\rangle \overset{*}{\Rightarrow} \langle v,\, \overline{\theta}_1,\, l\rangle$. By the corollary of Lemma 6, $l = l_I$. So the resulting state is $\langle v,\, \theta_2,\, l_I\rangle$ for some θ_2.

□

Theorem 2 *If $e \Downarrow v$ in λ_{c1cc}, where e is a user-written program, then we know that* (**let** *call1cc* $= \cdots$ **in** e) $\Downarrow v$ *in λ_{sym}. Moreover, if e diverges in λ_{c1cc}, then* **let** *call1cc* $= \cdots$ **in** e *diverges in λ_{sym}.*

Proof: If we follow the first steps of the evaluation of **let** *call1cc* $= \cdots$ **in** e, which evaluate the outer let, we have

$$\langle \mathbf{let}\; call1cc = \cdots\; \mathbf{in}\; e,\, \theta_0,\, l_I\rangle \overset{*}{\Rightarrow} \langle e,\, \theta_0[call1cc \leftarrow \ldots],\, l_I\rangle$$

If $e \Downarrow v$ in λ_{c1cc}, then $\langle e,\, \theta_0\rangle \overset{*}{\Rightarrow} \langle v,\, \theta_1\rangle$, for some store θ_1. By Lemma 7, this means that $\langle e,\, \theta_0[call1cc \leftarrow \ldots],\, l_I\rangle \overset{*}{\Rightarrow} \langle v,\, \theta_2,\, l_I\rangle$. Therefore,

$$\langle \mathbf{let}\; call1cc = \cdots\; \mathbf{in}\; e,\, \theta_0,\, l_I\rangle \overset{*}{\Rightarrow} \langle e,\, \theta_0[call1cc \leftarrow \ldots],\, l_I\rangle \overset{*}{\Rightarrow} \langle v,\, \theta_2,\, l_I\rangle$$

For the case where e diverges, the proof is similar to that of Theorem 1.

$\square$

3.3 One-shot Subcontinuations

Subcontinuations [HDA94] 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 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.

We can describe the semantics of subcontinuations 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.

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

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}:

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

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

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

The semantics of subcontinuations is described by the rules shown next:[7]

$$\begin{array}{rcll} \langle C[\textbf{spawn}\ v],\, \theta\rangle & \Rightarrow & \langle C[l{:}\, v\ (\lambda x.\, \textbf{controller}\ l\ x)],\, \theta\rangle & (20) \\ & & \text{where } l \text{ is a fresh label} & \\ \langle C[l{:}\, v],\, \theta\rangle & \Rightarrow & \langle C[v],\, \theta\rangle & (21) \\ \langle C_1[l{:}\, C_2[\textbf{controller}\ l\ v]],\, \theta\rangle & \Rightarrow & \langle C_1[v\ (\lambda x.\, l{:}\, C_2[x])],\, \theta\rangle & (22) \end{array}$$

Rule 20 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 21 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.

[7]Except for some syntactical adaptations, this is the operational semantics of subcontinuations developed in [HDA94].

Rule 22 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. 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 **subcont** l is treated as a value:

$$v \;\rightarrow\; \ldots \;\mid\; \textbf{subcont}\; l$$

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

$$\begin{aligned}
\langle C[\textbf{spawn}\; v],\, \theta\rangle &\;\Rightarrow\; \langle C[l\colon v\; (\lambda x.\, \textbf{controller}\; l\; x)],\, \theta[l \leftarrow \textbf{nil}]\rangle && (23)\\
&\qquad \text{where } l \text{ is a fresh label} \\
\langle C[l\colon v],\, \theta\rangle &\;\Rightarrow\; \langle C[v],\, \theta\rangle && (24)\\
\langle C_1[l\colon C_2[\textbf{controller}\; l\; v]],\, \theta\rangle &\;\Rightarrow\; \langle C_1[v\; (\textbf{subcont}\; l)],\, \theta[l \leftarrow \lambda x.\, C_2[x]]\rangle && (25)\\
\langle C[\textbf{subcont}\; l\; v],\, \theta\rangle &\;\Rightarrow\; \langle C[l\colon \theta(l)\; v],\, \theta[l \leftarrow \textbf{nil}]\rangle && (26)
\end{aligned}$$

Rules 23 and 24 are similar to Rules 20 and 21; the only difference is that Rule 23 clarifies the notion of a fresh label by stating that a fresh label is not present in the store.

According to Rule 25, 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 that subcontinuation.

Rule 26 shows that the invocation of a subcontinuation invalidates the mapping of its corresponding label; this prevents a subcontinuation to be shot more than once.

When we compare the semantics of one-shot subcontinuations with the semantics of full asymmetric coroutines (described in Section 2.2), 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 25) is very much like suspending an asymmetric coroutine (Rule 11). Invoking a one-shot subcontinuation (Rule 25) is also similar to resuming an asymmetric coroutine (Rule 10).

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

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

let $x = \Box$ **in** $controller{:}=$ **nil**; $\lambda y.\ x$

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

let $x = \Box$ **in** $subcont{:}=$ **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 those 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

```
let $x = \Box$ in
        if $controller$ then $reinstate(invokecontroller(x))$ else $x(reinstate)$
```

We will not provide a thorough proof of the correctness of our definition for *spawn*, both because it is lengthy and because it follows the same pattern of our previous proofs. Instead, we present next a sketch of that proof.

In our simulation of one-shot subcontinuations in λ_a, the auxiliary contexts C_r and C_t implement the actions that need to be performed when a simulated subcomputation suspends its execution and when it terminates. Therefore, when a subcomputation is active, we have the following relation between an evaluation context in λ_{subc} and in our simulation in λ_a:

$$C[l\colon e] \text{ in } \lambda_{subc} \cong C[C_r[l\colon C_t[e]]] \text{ in } \lambda_a$$

Let us trace the execution of *spawn* in our simulation. It produces the following steps:

$\langle C[spawn\, f],\, \theta\rangle \overset{*}{\Rightarrow}$
$\langle C[C_r[\textbf{resume}\ l_i\ (\lambda h.\ controller_i\ h)]],$
$\quad \theta[l_i \leftarrow \lambda c.\ C_t[f(c)],\, subcomp_i \leftarrow l_i,\, controller_i \leftarrow invokecontroller_i, \ldots]\rangle \overset{*}{\Rightarrow}$
$\langle C[C_r[l_i\colon C_t[f(\lambda h.\ controller_i\ h)]]],\ \theta[l_i \leftarrow \textbf{nil}, \ldots]\rangle$

We can observe that our simulation of **spawn** implements the semantics defined by Rule 23. Moreover, if we trace the termination of the simulated subcomputation, we observe that it also follows the semantics defined by Rule 24:

$$\langle C[C_r[l_i\colon C_t[v]]],\, \theta\rangle \overset{*}{\Rightarrow} \langle C[(\lambda x.\, v)(\mathit{reinstate}_i)],\, \theta\rangle \Rightarrow \langle C[v],\, \theta\rangle$$

For a controller invocation, and the subsequent invocation of its captured subcontinuation, we have two cases: the suspension and reactivation of a single subcomputation and the suspension and reactivation of an arbitrary number of nested subcomputations. We will show next that for both cases our simulation follows the semantics defined by Rules 25 and 26.

Let us first trace a controller invocation that involves a single subcomputation:

$$\begin{aligned}
&\langle C_1[C_r[l_i\colon C_t[C_2[\mathit{controller}_i\ g]]]],\, \theta\rangle \overset{*}{\Rightarrow}\\
&\langle C_1[C_r[l_i\colon C_t[C_2[C_y[\mathbf{yield}\ (\lambda x.\, g(x))]]]]],\\
&\qquad \theta[\mathit{subcont}_i \leftarrow \lambda x.\, \mathbf{resume}\ l_i\ x,\ \mathit{controller}_i \leftarrow \mathbf{nil}]\rangle \overset{*}{\Rightarrow}\\
&\langle C_1[C_r[\lambda x.\, g(x)]],\, \theta[l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\ \mathit{controller}_i \leftarrow \mathbf{nil},\, \ldots]\rangle \overset{*}{\Rightarrow}\\
&\langle C_1[g(\mathit{reinstate}_i)],\, \theta[l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\, \ldots]\rangle
\end{aligned}$$

It is easy to see that this corresponds to Rule 25. If the captured subcontinuation is invoked, our simulation reinstates the subcomputation, following the semantics defined by Rule 26:

$$\begin{aligned}
&\langle C[\mathit{reinstate}_i\ v],\, \theta[l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\ \mathit{subcont}_i \leftarrow \lambda x.\, \mathbf{resume}\ l_i\ x,\, \ldots]\rangle \overset{*}{\Rightarrow}\\
&\langle C[C_r[\mathit{subcont}_i\ v]],\, \theta[\mathit{controller}_i \leftarrow \mathit{invokecontroller}_i,\, \ldots]\rangle \overset{*}{\Rightarrow}\\
&\langle C[C_r[l_i\colon C_t[C_2[v]]]],\, \theta[\mathit{subcont}_i \leftarrow \mathbf{nil},\ l_i \leftarrow \mathbf{nil},\, \ldots]\rangle
\end{aligned}$$

Let us now trace a controller invocation from within a nested subcomputation. For the sake of simplicity, we will consider only one level of nesting; it is not difficult to generalize the argument to an arbitrary number of levels. We can observe that our simulation reaches the invoked controller's root, including the two subcomputations in the captured subcontinuation:

$$\begin{aligned}
&\langle C_o[C_r[l_o\colon C_t[C_i[C_r[l_i\colon C_t[C_2[\mathit{controller}_o\ g]]]]]]],\, \theta\rangle \overset{*}{\Rightarrow}\\
&\langle C_o[C_r[l_o\colon C_t[C_i[C_r[\lambda x.\, g(x)]]]]],\\
&\quad \theta[l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\ \mathit{controller}_o \leftarrow\ \mathbf{nil},\ \mathit{subcont}_o \leftarrow \lambda x.\, \mathbf{resume}\ l_o\ x]\rangle \overset{*}{\Rightarrow}\\
&\langle C_o[C_r[l_o\colon C_t[C_i[\mathit{reinstate}_i(\mathit{invokecontroller}_i(\lambda x.\, g(x)))]]]],\\
&\quad \theta[l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\ \mathit{controller}_o \leftarrow\ \mathbf{nil},\, \ldots]\rangle \overset{*}{\Rightarrow}\\
&\langle C_o[C_r[\lambda x.\, g(x)]],\\
&\quad \theta[l_o \leftarrow \lambda x.\, C_t[C_i[\mathit{reinstate}_i[C_y[x]]]],\ \mathit{controller}_i \leftarrow \mathbf{nil},\\
&\qquad \mathit{subcont}_i \leftarrow \lambda x.\, \mathbf{resume}\ l_i\ x, l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\, \ldots]\rangle \overset{*}{\Rightarrow}\\
&\langle C_o[g(\mathit{reinstate}_o)],\\
&\quad \theta[l_o \leftarrow \lambda x.\, C_t[C_i[\mathit{reinstate}_i[C_y[x]]]],\ l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\, \ldots]\rangle
\end{aligned}$$

Finally, if that subcontinuation is invoked, our simulation successfully reinstates the two nested subcomputations:

$$\begin{aligned}
&\langle C[\mathit{reinstate}_o\ v],\\
&\quad \theta[l_o \leftarrow \lambda x.\, C_t[C_i[\mathit{reinstate}_i[C_y[x]]]],\ \mathit{subcont}_o \leftarrow \lambda x.\, \mathbf{resume}\ l_o\ x,\\
&\qquad l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\ \mathit{subcont}_i \leftarrow \lambda x.\, \mathbf{resume}\ l_i\ x,\, \ldots]\rangle \overset{*}{\Rightarrow}\\
&\langle C[C_r[l_o\colon C_t[C_i[\mathit{reinstate}_i\ v]]]],\\
&\quad \theta[l_i \leftarrow \lambda x.\, C_t[C_2[C_y[x]]],\ \mathit{subcont}_i \leftarrow \lambda x.\, \mathbf{resume}\ l_i\ x,\\
&\qquad \mathit{subcont}_o \leftarrow \mathbf{nil}, l_o \leftarrow \mathbf{nil},\, \ldots]\rangle \overset{*}{\Rightarrow}\\
&\langle C[C_r[l_o\colon C_t[C_i[C_r[l_i\colon C_t[C_2[v]]]]]]],\\
&\quad \theta[\mathit{subcont}_i \leftarrow \mathbf{nil}, l_i \leftarrow \mathbf{nil},\ \mathit{subcont}_o \leftarrow \mathbf{nil}, l_o \leftarrow \mathbf{nil},\, \ldots]\rangle
\end{aligned}$$

4 Final Remarks

In this paper, 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.

It is interesting to compare our notion of expressive power with Felleisen's [Fel90]. 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).

The next step is to explore negative results, such as a formal proof that we cannot express full coroutines using restricted forms of coroutines, or multi-shot 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 it. But we still have to formalize these ideas.

References

[BWD96] C. Bruggeman, O. Waddell, and R. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN'96 Conf. on Programming Language Design and Implementation (PLDI'96)*, pages 99–107, Philadelphia, PA, May 1996. ACM. SIGPLAN Notices 31(5).

[dMI09] Ana Lucia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 2009. to appear.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In *Proceedings of 3rd European Symposium on Programming ESOP'90*, pages 134–151, Copenhagen, Denmark, May 1990.

[FF86] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts-III*, pages 193–217. North-Holland, 1986.

[HDA94] R. Hieb, R. Dybvig, and Claude W. Anderson III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[HF87] C. T. Haynes and D. P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Progamming Languages and Systems*, 9(4):582–598, October 1987.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, La Jolla, CA, 1995.