Typing Coroutines

Konrad Anton and Peter Thiemann

Institut für Informatik, Universität Freiburg, Germany
{anton,thiemann}@informatik.uni-freiburg.de

Abstract. A coroutine is a programming construct between function and thread. It behaves like a function that can suspend itself arbitrarily often to yield intermediate results and to get new inputs before returning a result. This facility makes coroutines suitable for implementing generator abstractions.

Languages that support coroutines are often untyped or they use trivial types for coroutines. This work supplies the first type system with dedicated support for coroutines. The type system is based on the simply-typed lambda calculus extended with effects that describe control transfers between coroutines.

1 Introduction

A coroutine is a programming construct between function and thread. It can be invoked like a function, but before it returns a value (if ever) it may suspend itself arbitrarily often to return intermediate results and then be resumed with new inputs. Unlike with preemptive threading, a coroutine does not run concurrently with the rest of the program, but rather takes control until it voluntarily suspends to either return control to its caller or to pass control to another coroutine. Coroutines are closely related to cooperative threading, but they add value because they are capable of passing values into and out of the coroutine and they permit explicit switching of control.

Coroutines were invented in the 1960s as a means for structuring a compiler [4]. They have received a lot of attention in the programming community and have been integrated into a number of programming languages, for instance in Simula 67 [5], BETA, CLU [11], Modula-2 [19], Python [17], and Lua [15], and Knuth finds them convenient in the description of algorithms [8]. Coroutines are also straightforward to implement in languages that offer first-class continuations (e.g., Scheme [7]) or direct manipulation of the execution stack (e.g., assembly language, Smalltalk).

The main uses of coroutines are the implementation of compositions of state machines as in Conway's seminal paper [4] and the implementation of generators. A generator enumerates a potentially infinite set of values with successive invocations. The latter use has led to renewed interest in coroutines and to their inclusion in mainstream languages like C# [13], albeit in restricted form as generators.

R. Page, Z. Horváth, and V. Zsók (Eds.): TFP 2010, LNCS 6546, pp. 16–30, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Despite the renewed interest in the programming construct per se, the typing aspects of coroutines have not received much attention. Indeed, the supporting languages are either untyped (e.g., Lua, Scheme, Python), the typing for coroutines is trivialized, or coroutines are restricted so that a very simple typing is sufficient. For instance, in Modula-2, coroutines are created from parameterless procedures so that all communication between coroutines must take place through global variables. Also, for describing generators, a simple function type seems sufficient.

Contribution. We propose a static type system for first-class, stackful coroutines that may be used in both, symmetric and asymmetric ways.[1] Moreover, we permit passing arguments to a coroutine at each start and resume operation, and we permit returning results on each suspend and on termination of the coroutine (and we distinguish between these two events). Our type system is based on the simply-typed lambda calculus. It includes an effect system that describes the way the coroutine operations are used. We present a small-step operational semantics for the language and prove type soundness.

Outline. Sec. 2 describes the language CorDuroy. It starts with some examples (Sec. 2.1) before delving into operational semantics (Sec. 2.2) and the type system (Sec. 2.3). Sec. 3 proves type soundness by establishing preservation and progress properties following the syntactic approach [20]. Sec. 4 discusses related work, and Sec. 5 concludes and outlines directions of further research.

2 CorDuroy

The language CorDuroy is a simply typed lambda calculus with recursive functions and operations for handling coroutines. Fig. 1 specifies the syntax; labels ℓ only occur at run time. We define λ-abstraction as sugar for the fixpoint operator: $\lambda x.e := \mathsf{fix}\,\lambda\_.\lambda x.e$.

Coroutines in CorDuroy are run-time entities identified by a label ℓ. The only way to create them is by applying the create operator to a function. Once a coroutine has been created, it can be executed. Unlike threads in a multi-threaded language, only one coroutine is active at any given time.

To activate a coroutine, there is a symmetric (transfer) and an asymmetric (resume) operator. The symmetric operator transfer suspends the currently executing coroutine and executes another[2]. The asymmetric operator resume builds

[1] This terminology is due to De Moura and Ierusalemschy [14]. A coroutine is *stackful*, if it can suspend inside nested function calls. Coroutines are *asymmetric* if coroutine activity is organized in a tree-like manner: each coroutine invocation or resumption always returns and yields to its caller. In contrast, *symmetric* coroutines can transfer control among each others without restrictions.

[2] We use the keywords established by De Moura and Ierusalemschy [14]. In Simula [5], transfer corresponds to the system procedure `RESUME`, whereas "asymmetric", yield and resume correspond to "semi-symmetric", `DETACH` and `CALL`, respectively.

$$
\begin{aligned}
B &::= \mathsf{Bool} \mid \mathsf{Unit} \mid \dots & v &::= k^0 \mid \mathsf{fix}\,\lambda f.\lambda x.e \mid \ell \\
k^0 &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid \dots & e &::= k^n\, e_1 \dots e_n \mid \mathsf{fix}\,\lambda f.\lambda x.e \\
k^1 &::= \neg \mid \dots & &\quad \mid x \mid e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \\
k^2 &::= \wedge \mid \dots & &\quad \mid \mathsf{create}\, x.e \mid \mathsf{yield}\, e \\
\ell, \ell', \dots &\in \mathrm{Labels} & &\quad \mid \mathsf{resume}\, e\, e\, e\, e \mid \mathsf{transfer}\, e\, e \\
x, y, f, \dots &\in \mathrm{Var} & &\quad \mid \ell \\
& & \varphi &::= \bot \mid \tau \ddot{\rightsquigarrow} \tau / \tau \mid \top \\
& & \tau &::= B \mid \tau \xrightarrow{\varphi} \tau \mid \top \mid \bot \mid \tau \rightsquigarrow \tau / \tau
\end{aligned}
$$

Fig. 1. Syntax

a caller-callee relationship: if a coroutine resumes another coroutine, they become caller and callee. The yield operator inside the callee suspends the coroutine and returns control to the caller. Each of the three operators passes a value. In the remaining paper, we understand "activate" to mean either transfer or resume, but not yield.

The caller-callee relationship is also used when a coroutine finally returns a value, as the value is then passed to the caller. Activating a coroutine after it has returned causes a run-time error; hence, the caller needs to know whether the callee coroutine has terminated. resume requires therefore as its third and fourth parameter two *result functions*, one to call with yielded values and one to call with the returned value[3].

The language includes a countable set of primitive functions k^n, having each an arity $n \geq 0$. Partial application of primitive functions is not allowed.

2.1 Examples

This section contains short examples of CorDuroy programs. We assume that integers and strings are among the basic types B and that there are constants k^n for arithmetic operations, comparison, and printing. We also use the common $\mathsf{let} \cdot = \cdot \,\mathsf{in}\, \cdot$ sugar for readability.

Divisors. Generators can be used to compute sequences one element at a time. Fig. 2(a) shows a coroutine which generates the divisors of a number, and a consumer which iterates over the divisors until the generator returns (and the second result function of resume is called).

Mutable references. Coroutines are the only stateful construct in CorDuroy. In Fig. 2(b), a mutable reference is simulated by a coroutine which keeps an integer value in a local variable. Whenever it is resumed with a function $\mathsf{Int} \to \mathsf{Int}$, it

[3] Alternatively, the λ-calculus could be extended with variant types in order to tag the result of resume with how it was obtained. We chose the two-continuation resume for simplicity.

```
let divisors_of = λn.
  create _.λ_.
    ((fix λloop.λk.
      if (> k n) then unit
      else let rem = (mod n k) in
        let _= (if (= rem 0)
          then yield k else unit)
        in loop (+ k 1 )) 1)
in let g = divisors_of 24 in
  ((fix λf. λ_.
    resume g unit
      (λn. let _ = (print_int n)
          in (f unit) )
      (λ_. (print_str "finito")))
    unit)
// output: 1 2 3 4 6 8 12 24 finito
```

(a) Compute all divisors.

```
let makeref = λx0.
  let main = fix λloop.λx.λupd.
      let x' = upd x in
      let upd' = yield x' in
        loop x' upd'
  in create _. main x0
in let undef = fix λf.λx.(f x)
in let write = λr.λv.
  resume r (λ_.v)
    (λ_.unit) (λ_.unit)
in let read = λ r.
  resume r (λx.x) (λx.x) undef
in
 let r = makeref 1 in
 let _ = print_int (read r) in
 let _ = write r 2 in
 print_int (read r)
// output: 1 2
```

(b) Mutable references.

Fig. 2. Code examples

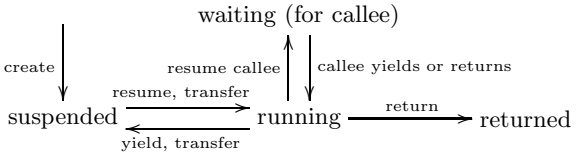

Fig. 3. Life cycle of coroutines

lets the function update the value and returns the new value. The example also shows how fix can be used to create a diverging function with any desired return type in the read function[4].

2.2 Operational Semantics

This section presents a small-step operational semantics for CorDuroy, starting with a life-cycle based view on coroutines to motivate the stack-based representation used in the reduction rules in Fig. 5.

The life cycle of a coroutine consists of the states *suspended*, *running*, *waiting* and *returned*, as shown in Fig. 3. At any moment, there is only one running coroutine.

The running coroutine can apply create to a function, creating a new coroutine which starts life in the suspended state (E-Create). It can also resume a

[4] The language could alternatively be extended with a special variant of the resume operator for coroutines which never return.

$$
\begin{aligned}
C ::= {} & \Box \mid k^n\, v_1 \ldots v_{i-1}\ C\ e_{i+1} \ldots e_n \quad (1 \le i \le n) \\
& \mid e\, C \mid C\, v \mid \mathsf{if}\, C\, \mathsf{then}\, e\, \mathsf{else}\, e \\
& \mid \mathsf{resume}\, C\, e\, e\, e \mid \mathsf{resume}\, v\, C\, e\, e \mid \mathsf{resume}\, v\, v\, C\, e \mid \mathsf{resume}\, v\, v\, v\, C \\
& \mid \mathsf{yield}\, C \mid \mathsf{transfer}\, C\, e \mid \mathsf{transfer}\, v\, C
\end{aligned}
$$

$$
\begin{aligned}
S &::= \ell @ e; S^? & \mathrm{labels}(\ell @ e; S^?) &= \{\ell\} \cup \mathrm{labels}(S^?) \\
S^? &::= \epsilon \mid S & \mathrm{labels}(\epsilon) &= \emptyset
\end{aligned}
$$

Fig. 4. Evaluation contexts and stacks

$$\frac{n > 0}{\left\langle \ell @ C[k^n\, v_1 \ldots v_n]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell @ C[[\![k^n]\!]\,(v_1, \ldots, v_n)]\,;\, S^? \mid \mu \right\rangle}\ \text{E-Const}$$

$$\frac{}{\left\langle \ell @ C[(\mathsf{fix}\, \lambda f.\lambda x.e)\, v]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell @ C[e[f \mapsto \mathsf{fix}\, \lambda f.\lambda x.e]\,[x \mapsto v]\,]\,;\, S^? \mid \mu \right\rangle}\ \text{E-Fix}$$

$$\frac{}{\left\langle \ell @ C[\mathsf{if}\, \mathsf{true}\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell @ C[e_t]\,;\, S^? \mid \mu \right\rangle}\ \text{E-IfT}$$

$$\frac{}{\left\langle \ell @ C[\mathsf{if}\, \mathsf{false}\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell @ C[e_f]\,;\, S^? \mid \mu \right\rangle}\ \text{E-IfF}$$

$$\frac{\begin{array}{c} x^* \notin \mathrm{free}(e) \cup \{x\} \qquad \ell^* \notin \mathrm{dom}\,(\mu) \cup \mathrm{labels}(S^?) \cup \{\ell\} \\ v^* = \lambda x^*.((\lambda x.e\, x^*)\, \ell^*) \qquad \mu' = \mu \cup \{(\ell^*, v^*)\} \end{array}}{\left\langle \ell @ C[\mathsf{create}\, x.e]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell @ C[\ell^*]\,;\, S^? \mid \mu' \right\rangle}\ \text{E-Create}$$

$$\frac{\ell' \in \mathrm{dom}\,(\mu) \qquad e = \mathsf{resume}\, \ell'\, v_a\, v_s\, v_n}{\left\langle \ell @ C\,[e]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell' @\, (\mu\,(\ell')\ v_a)\,;\, \ell @ C\,[e]\,;\, S^? \mid \mu \setminus \ell' \right\rangle}\ \text{E-Res}$$

$$\frac{x^* \text{ fresh} \qquad e_2 = \mathsf{resume}\, \ell\, v_a\, v_s\, v_n}{\left\langle \ell_1 @ C_1\,[\mathsf{yield}\, v_y]\,;\, \ell_2 @ C_2\,[e_2]\,;\, S^? | \mu \right\rangle \to \left\langle \ell_2 @ C_2\,[v_s\ v_y]\,;\, S^? | \mu\,[\ell_1 \mapsto \lambda x^*.C_1[x^*]] \right\rangle}\ \text{E-Yie}$$

$$\frac{}{\left\langle \ell_1 @ v_r; \ell_2 @ C_2\,[\mathsf{resume}\, \ell\, v_a\, v_s\, v_n]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell_2 @ C_2\,[v_n\ v_r]\,;\, S^? \mid \mu \right\rangle}\ \text{E-CoRet}$$

$$\frac{}{\left\langle \ell @ C\,[\mathsf{transfer}\, \ell\, v_a]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell @ C\,[v_a]\,;\, S^? \mid \mu \right\rangle}\ \text{E-TraSelf}$$

$$\frac{\ell' \in \mathrm{dom}\,(\mu) \qquad x^* \text{ fresh}}{\left\langle \ell @ C\,\left[\mathsf{transfer}\, \ell'\, v\right]\,;\, S^? \mid \mu \right\rangle \to \left\langle \ell' @ (\mu(\ell')\, v)\,;\, S^? \mid (\mu \setminus \ell') \cup \{(\ell, \lambda x^*.C[x^*])\} \right\rangle}\ \text{E-Tra}$$

$$\text{E-TraErr}\ \frac{\ell' \notin \mathrm{dom}\,(\mu)}{\left\langle \ell @ C\,\left[\mathsf{transfer}\, \ell'\, v_a\right] \mid \mu \right\rangle \to \mathsf{Error}} \qquad \text{E-ResErr}\ \frac{\ell' \notin \mathrm{dom}\,(\mu)}{\left\langle \ell @ C\,\left[\mathsf{resume}\, \ell'\, v_a\, v_s\, v_n\right] \mid \mu \right\rangle \to \mathsf{Error}}$$

Fig. 5. Small-step operational semantics rules

suspended coroutine (the *callee*), becoming its *caller* (E-Res). In doing so, it enters the *waiting* state, and the callee becomes *running*.

A running coroutine can also yield, after which it is *suspended* and the caller *running* (E-Yie). If a running coroutine reduces to a value, it is said to *return* that value. The returning coroutine enters its terminal state, and the value is then passed to the caller if there is one (E-CoRet) or becomes the final result of the program.

Alternatively, the running coroutine can transfer control to a suspended coroutine, suspending itself. In this case, the successor coroutine not only enters the running state, but it also becomes the (only) callee of the predecessor's caller (E-Tra).

In the rules, the state of a program being evaluated is represented as a pair $\langle S \mid \mu \rangle$ of stack and store. The stack S contains, from left to right, the running coroutine, its caller, its caller's caller and so on, each in the form of labeled contexts $\ell@e$ (see Fig. 4). As the running coroutine is the top of the stack, the reduction rules must never pop the last labeled context off the stack.

All suspended coroutines[5] are kept in the store μ, a function from labels ℓ to values v. The values in the store are the continuations of the yield and transfer expressions which caused the coroutine to be suspended, or, in the case of newly created coroutines, functions which are constructed to be applied likewise.

Coroutines in the returned state are neither in the stack nor in the store because they play no further role in the execution. The waiting and suspended states resemble each other in that neither state permits β-reductions.

The coroutine-related rules all maintain the invariant that a coroutine never rests in the stack and in the store simultaneously. Rule E-Create sets up a continuation v^* which makes the new label ℓ^* known under the name x inside the body expression e and passes the first input value to e. E-Res removes the stored continuation of the given coroutine from the store and applies it to the argument in a new labeled context on top of the stack. In the now-waiting coroutine, the resume expression remains, awaiting a result from the coroutine above. The third and fourth resume parameters are the result functions to be called later with yielded and returned values, respectively.

E-Yie and E-Tra put the continuation of the running coroutine into the store. While E-Yie passes the argument to the first of the two result functions of the caller, E-Tra sets up a new stack top in which the continuation from the store is applied to the argument, just like in E-Res. E-CoRet passes the return value to the other result function in the caller and discards the callee. Of the resume expression in E-Yie and E-CoRet, only the two result functions are used; the old label ℓ need not match the returning or yielding coroutine because the stack top may have been replaced in a transfer action.

If a coroutine attempts to activate another coroutine which is not in the store (i.e., not suspended), execution aborts with a run-time error (E-ResErr,

[5] An implementation would keep the coroutines within the store all the time and annotate them with their state instead; however, the notion of putting coroutines into the store and taking them out again makes the rules easier to read.

E-TraErr)[6]. As an exception to this rule, a coroutine may safely transfer to itself, e.g. in a multitasking system with just one ready task (E-TraSelf).

Rule E-Yie is only enabled if the stack contains a suitable waiting coroutine below; fortunately, the type system rejects all programs in which a yield expression could appear as a redex in the lowest labeled context.

There is no distinguished main program; the initial expression is also treated as a coroutine, except that it starts in the running state. In order to evaluate a CorDuroy program e, it is wrapped in an initial state with the fixed label ℓ_0:

$$\text{initState}(e) = \langle S_0 \mid \emptyset \rangle \quad S_0 = \ell_0 @ e; \epsilon \tag{1}$$

The function $[\![\cdot]\!]$ in E-Const maps primitive function symbols $k^n, n > 0$ to partial functions of the same arity. The notation $e[x \mapsto f]$ stands for standard capture-avoiding substitution which replaces all free occurrences of x in e by f. The set of free variables in e is $\text{free}(e)$.

2.3 Type System

The type system ensures that values passed to and from coroutines do not cause type errors at run time, and that coroutine operations within the same coroutine body are compatible with each other. It is based on the simply-typed λ-calculus, with an effect system describing which coroutine actions may occur during the evaluation of an expression.

Effects. The effect part of the type and effect system summarizes the yield and transfer expressions which may be evaluated during the evaluation of an expression. The propagation of effects through function application permits a called function to yield and transfer on behalf of the running coroutine in a type-safe way.

If an expression has the effect $\tau_i \ddot{\rightsquigarrow} \tau_o / \tau_r$, then its execution may yield a value of type τ_o to the calling coroutine and expect a value of type τ_i when it is activated again. It may also transfer execution to a coroutine which yields values of type τ_o or returns a value of type τ_r.

Effects φ form a lattice with bottom element $\bot$ and top element $\top$ (see Fig. 6). $\bot$ means that the expression will under no circumstance ever yield. Effect $\top$ means that yield expressions with different types are possible and nothing can be said about the values.

Types. The type system features basic types B, function types, coroutine types as well as top and bottom types.

[6] This class of runtime errors can be eliminated if E-Res and E-Tra leave the coroutine in the store. Then, activating a terminated or waiting coroutine would invoke (a copy of) the last stored continuation, similar to multi-shot continuations. We chose the error-rules because they are more similar to how Lua and Python handle these situations, and they do not need a facility to copy continuations.

$$\bot \sqcup \varphi = \varphi \sqcup \bot = \varphi$$
$$\top \sqcup \varphi = \varphi \sqcup \top = \top$$

$$(\tau_i \ddot{\rightsquigarrow} \tau_o / \tau_r) \sqcup (\tau_i' \ddot{\rightsquigarrow} \tau_o' / \tau_r') = (\tau_i \sqcap \tau_i') \ddot{\rightsquigarrow} (\tau_o \sqcup \tau_o') \,/\, (\tau_r \sqcup \tau_r')$$

$$\varphi_1 \sqsubseteq \varphi_2 \quad \text{iff} \quad \exists \varphi_1'. \varphi_1 \sqcup \varphi_1' = \varphi_2$$

(a) Effects.

$$\top \sqcup \tau = \tau \sqcup \top = \top$$
$$\bot \sqcap \tau = \tau \sqcap \bot = \bot$$
$$\bot \sqcup \tau = \tau \sqcup \bot = \top \sqcap \tau = \tau \sqcap \top = \tau$$

$$\tau_1^+ \sqcup \tau_2^+ = \begin{cases} \tau_1^+ & \tau_1^+ = \tau_2^+ \\ \top & \text{otherwise} \end{cases}$$

$$\tau_1^+ \sqcap \tau_2^+ = \begin{cases} \tau_1^+ & \tau_1^+ = \tau_2^+ \\ \bot & \text{otherwise} \end{cases}$$

(b) Types.

Fig. 6. Join and meet

Function arrows are annotated with the effect which may occur during the function's evaluation. We write $\tau_1 \to \tau_2$ for $\tau_1 \xrightarrow{\bot} \tau_2$.

A value of type $\tau_i \rightsquigarrow \tau_o / \tau_r$ corresponds to a coroutine which can be resumed with values of input type τ_i and yields values of output type τ_o or returns a value of return type τ_r.

Types form a flat lattice with bottom $\bot$ and top $\top$. For simplicity, subtyping is not allowed, and subeffecting is only allowed in create and fix expressions. Join and meet on types are defined in figure 6, where τ^+ represent types except for $\top$ and $\bot$.

Typing rules. The rules are given in Fig. 7. The type environment Γ maps variables to their types. The store typing

$$\Sigma \subseteq \text{Labels} \times \{\tau_i \rightsquigarrow \tau_o / \tau_r | \tau_{i,o,r} \neq \top, \tau_i \neq \bot\} \qquad (\text{Def}\Sigma)$$

maps labels to the types of the corresponding coroutines at run time. The exclusion of $\top$ and $\bot$ serves to avoid subtyping. Note that type rules do not extend Σ; expressions are type-checked against a fixed Σ, and preservation (Sec. 3.1) guarantees that some Σ can be found after each evaluation step.

The type function $\text{basty}_\text{k}(k^n)$ maps constants to their types of the form $B_1 \to B_2 \to \ldots \to B_{n+1}$. We assume that $\text{basty}_\text{k}(k^n)$ agrees with the primitive denotation $[\![k^n]\!]$. We also assume that true and false are the only k^0 of type Bool, and that only unit inhabits Unit.

Most type rules compute the effect of their expression by joining the effects of the subexpressions. The only exceptions are T-Fix and T-Create, in which the effect of the body expression is moved onto the function arrow or into the coroutine type.

The create expression creates a coroutine from a function. In doing so, it binds a variable to the freshly created coroutine label.

yield and transfer contribute an effect with its input type τ_i. Both suspend the current coroutine and expect a value of type τ_i the next time it is activated. The output and return types in the effect of yield describe that yield certainly causes

the coroutine to yield a value of that type, but never causes a return. transfer, however, transfers control *and the relationship to the caller* to a coroutine which, in turn, may yield and return. Therefore, T-Tra puts the other coroutine's output and return types into the effect in order to force the surrounding yield and return expressions to match.

Rule T-Prog defines when an entire program is well-typed. The input type Unit is an arbitrary choice, but since the initial label ℓ_0 is not lexically accessible in the program, the input type is of little importance anyway[7]. The output type is bounded to $\bot$ so that an expression which yields can never be the bottom-most expression in a stack (and yield with $e : \bot$, while allowed, will diverge instead of yielding).

The initial store typing for a program e with $\vdash_{\text{prog}} e : \tau$ is defined as follows:

$$\Sigma_0^\tau = \{(\ell_0, \mathsf{Unit} \rightsquigarrow \bot/\tau)\} \tag{2}$$

3 Soundness

This section contains the soundness proof [8]. In Sec. 3.1, we prove that reduction steps preserve typing. Sec. 3.2 contains the progress proof, stating that all well-typed execution states are reducible or have finished.

3.1 Preservation

This section states and proves the preservation theorem (Theorem 1). We define the notion of a well-typed execution state before we formulate some lemmas in preparation for the main proof.

Fig. 8 contains the definition of an execution state $\langle S \mid \mu \rangle$ being well-typed, T-State. Apart from requiring that the types of store and stack members correspond to the store typing Σ, which is defined in T-Store and T-StackN, it poses a constraint $\Sigma \vdash_{\text{w}} S$ about the waiting coroutines in the stack: the redex of waiting callers must be a resume expression whose result functions are compatible with the output and return types of the callee.

Lemma 1. *If* $\emptyset|\Sigma \vdash C[e] : \tau\&\varphi$*, then* $\emptyset|\Sigma \vdash e : \tau'\&\varphi'$ *for some* τ'*,* $\varphi' \sqsubseteq \varphi$*, and* $\text{free}(e) = \emptyset$

Lemma 2. *If* $\Gamma|\Sigma \vdash v : \tau\&\varphi$*, then* $\varphi = \bot$ *and* $\tau \neq \bot$*.*

Lemma 3. *If* $\Gamma, x : \tau'|\Sigma \vdash e : \tau\&\varphi$ *and* $\emptyset|\Sigma \vdash v : \tau'\&\bot$*, then* $\Gamma|\Sigma \vdash e[x \mapsto v] : \tau\&\varphi$*.*

[7] If the program's design features multiple coroutines transferring to each other, there is still the possibility of having the initial program create one or more such coroutines, each of which knows its label, and transferring control to one of them.

[8] For space reasons, we have omitted most proofs. They are contained in the extended version of this paper, available from `http://proglang.informatik.uni-freiburg.de/projects/coroutines/`

$$\text{T-Const}\ \frac{\mathrm{basty_k}\,(k^n) = B_1 \to \ldots \to B_{n+1} \qquad \forall i = 1 \ldots n.\ \Gamma|\Sigma \vdash e_i : B_i \& \varphi_i}{\Gamma|\Sigma \vdash k^n\, e_1 \ldots e_n : B_{n+1} \& \bigsqcup_{i=1\ldots n} \varphi_i} \qquad \text{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma|\Sigma \vdash x : \tau \& \bot}$$

$$\text{T-App}\ \frac{\Gamma|\Sigma \vdash e_1 : \tau_2 \xrightarrow{\varphi_3} \tau_1 \& \varphi_1 \qquad \Gamma|\Sigma \vdash e_2 : \tau_2 \& \varphi_2}{\Gamma|\Sigma \vdash e_1\, e_2 : \tau_1 \& \varphi_1 \sqcup \varphi_2 \sqcup \varphi_3}$$

$$\text{T-If}\ \frac{\Gamma|\Sigma \vdash e_c : \mathsf{Bool} \,\& \varphi_c \qquad \Gamma|\Sigma \vdash e_t : \tau \& \varphi_t \qquad \Gamma|\Sigma \vdash e_f : \tau \& \varphi_f}{\Gamma|\Sigma \vdash \mathsf{if}\, e_c\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f : \tau \& \varphi_c \sqcup \varphi_t \sqcup \varphi_f}$$

$$\text{T-Fix}\ \frac{\Gamma, f : \tau_1 \xrightarrow{\varphi} \tau_2, x : \tau_1 | \Sigma \vdash e : \tau_2 \& \varphi' \qquad \varphi' \sqsubseteq \varphi}{\Gamma|\Sigma \vdash \mathsf{fix}\, \lambda f.\lambda x.e : (\tau_1 \xrightarrow{\varphi} \tau_2) \& \bot} \qquad \text{T-Label}\ \frac{\Sigma(\ell) = \tau_i \rightsquigarrow \tau_o / \tau_r}{\Gamma|\Sigma \vdash \ell : \tau_i \rightsquigarrow \tau_o / \tau_r \& \bot}$$

$$\text{T-Create}\ \frac{\Gamma, x : \tau_i \rightsquigarrow \tau_o / \tau_r | \Sigma \vdash e : \tau_i \xrightarrow{\varphi} \tau_r \& \varphi' \quad \varphi, \varphi' \sqsubseteq \tau_i \dddot{\rightsquigarrow} \tau_o / \tau_r \quad \tau_{i,o,r} \neq \top, \tau_i \neq \bot}{\Gamma|\Sigma \vdash \mathsf{create}\, x.e : \tau_i \rightsquigarrow \tau_o / \tau_r \& \bot}$$

$$\text{T-Res}\ \frac{\begin{array}{c}\Gamma|\Sigma \vdash e_c : \tau_i \rightsquigarrow \tau_o / \tau_r \& \varphi_1 \\ \Gamma|\Sigma \vdash e_a : \tau_i \& \varphi_2 \qquad \Gamma|\Sigma \vdash e_s : \tau_o \xrightarrow{\varphi_3} \tau_q \& \varphi_4 \qquad \Gamma|\Sigma \vdash e_n : \tau_r \xrightarrow{\varphi_5} \tau_q \& \varphi_6\end{array}}{\Gamma|\Sigma \vdash \mathsf{resume}\, e_c\, e_a\, e_s\, e_n : \tau_q \& \bigsqcup_{i=1\ldots6} \varphi_i}$$

$$\text{T-Yie}\ \frac{\Gamma|\Sigma \vdash e : \tau_o \& \varphi_1 \quad \tau_i \neq \top}{\Gamma|\Sigma \vdash \mathsf{yield}\, e : \tau_i \& (\tau_i \dddot{\rightsquigarrow} \tau_o / \bot) \sqcup \varphi_1} \qquad \text{T-Tra}\ \frac{\Gamma|\Sigma \vdash e_c : \tau_a \rightsquigarrow \tau_o / \tau_r \& \varphi_1 \qquad \Gamma|\Sigma \vdash e_a : \tau_a \& \varphi_2}{\Gamma|\Sigma \vdash \mathsf{transfer}\, e_c\, e_a : \tau_i \& (\tau_i \dddot{\rightsquigarrow} \tau_o / \tau_r) \sqcup (\varphi_1 \sqcup \varphi_2)}$$

$$\text{T-Prog}\ \frac{\emptyset|\emptyset \vdash e : \tau \& \varphi \quad \varphi \sqsubseteq \mathsf{Unit} \dddot{\rightsquigarrow} \bot / \tau}{\vdash_{\mathrm{prog}} e : \tau}$$

Fig. 7. Typing rules

Definition 1. *Given Γ, Σ, we write $\Gamma|\Sigma \vdash e_1 \leq e_2$, if $\Gamma|\Sigma \vdash e_1 : \tau \& \varphi_1$ and $\Gamma|\Sigma \vdash e_2 : \tau \& \varphi_2$ with $\varphi_1 \sqsubseteq \varphi_2$. $\Sigma \vdash e_1 \leq e_2$ is an abbreviation for $\emptyset|\Sigma \vdash e_1 \leq e_2$.*

Lemma 4 (Contexts are effect-monotone). *If $\Gamma|\Sigma \vdash e' \leq e$ and for some τ, φ, $\Gamma|\Sigma \vdash C[e] : \tau \& \varphi$, then $\Gamma|\Sigma \vdash C[e'] : \tau \& \varphi'$ for some $\varphi' \sqsubseteq \varphi$, and $\Gamma|\Sigma \vdash C[e'] \leq C[e]$.*

Lemma 5. *Let $S = \ell @ C[e]\,; S^?$, $S' = \ell @ C[e']\,; S^?$ such that $\Sigma \vdash e' \leq e$. Then $\Sigma \vdash_{le^*} S \Rightarrow \Sigma \vdash_{le^*} S'$ and $\Sigma \vdash_w S \Rightarrow \Sigma \vdash_w S'$ hold.*

Lemma 6. *Let $S = \ell_1 @ e_1; \ell_2 @ C[\mathsf{resume}\, \ell'\, v_p\, v_s\, v_n]\,; S^?$ such that $\Sigma \vdash_{le^*} S$ and $\Sigma \vdash_w S$. Let $\tau_i \rightsquigarrow \tau_s / \tau_n = \Sigma(\ell_1)$. Let v be a value with $\emptyset|\Sigma \vdash v : \tau_\alpha \& \bot$ for an $\alpha \in \{s, n\}$. Then, $S' = \ell_2 @ C[v_\alpha\, v]\,; S^?$ satisfies $\Sigma \vdash_{le^*} S'$ and $\Sigma \vdash_w S'$.*

$$\text{T-State}\quad \frac{\Sigma \vdash_{\text{sto}^*} \mu \qquad \Sigma \vdash_{\text{le}^*} S \qquad \Sigma \vdash_{\text{w}} S \qquad \text{labels}(S) \cap \text{dom}\,(\mu) = \emptyset}{\Sigma \vdash \langle S \mid \mu \rangle}$$

$$\text{T-Stack0}\quad \frac{}{\Sigma \vdash_{\text{le}^*} \epsilon} \qquad \text{T-StackN}\quad \frac{\ell \notin \text{labels}(S^?) \qquad \Sigma \vdash_{\text{le}} \ell@e \qquad \Sigma \vdash_{\text{le}^*} S^?}{\Sigma \vdash_{\text{le}^*} S \quad \text{where } S = \ell@e; S^?}$$

$$\text{T-StackE}\quad \frac{\Sigma(\ell) = \tau_i \rightsquigarrow \tau_o/\tau_r \qquad \emptyset|\Sigma \vdash e : \tau_r \& \varphi \qquad \varphi \sqsubseteq \tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r}{\Sigma \vdash_{\text{le}} \ell@e}$$

$$\text{T-WaitN}\quad \frac{S = \ell_1@e_1; S' \qquad \tau_i \rightsquigarrow \tau_o/\tau_r = \Sigma(\ell_1) \qquad S' = \ell_2@e_2; S^? \qquad e_2 = C[\mathsf{resume}\,\ell\,v_a\,v_s\,v_n] \qquad \emptyset|\Sigma \vdash v_s : \tau_o \xrightarrow{\varphi_s} \tau \& \bot \qquad \emptyset|\Sigma \vdash v_n : \tau_r \xrightarrow{\varphi_n} \tau \& \bot \qquad \Sigma \vdash_{\text{w}} S'}{\Sigma \vdash_{\text{w}} S}$$

$$\text{T-Wait1}\quad \frac{S = \ell@e; \epsilon \qquad \Sigma(\ell) = \tau_i \rightsquigarrow \bot/\tau_r}{\Sigma \vdash_{\text{w}} S} \qquad \text{T-Store}\quad \frac{\mu \text{ is function} \qquad \forall (\ell, v) \in \mu : \Sigma \vdash_{\text{sto}} (\ell, v)}{\Sigma \vdash_{\text{sto}^*} \mu}$$

$$\text{T-StoreE}\quad \frac{\Sigma(\ell) = \tau_i \rightsquigarrow \tau_o/\tau_r \qquad \emptyset|\Sigma \vdash v : \tau_i \xrightarrow{\varphi} \tau_r \& \bot \qquad \varphi \sqsubseteq \tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r}{\Sigma \vdash_{\text{sto}} (\ell, v)}$$

Fig. 8. Well-typed execution states, stacks, stores

Lemma 7. *If* $\tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r \sqsubseteq \tau_i' \ddot{\rightsquigarrow} \tau_o'/\tau_r'$*, then all of the following hold:*

- $\tau_i = \tau_i'$ *or* $\tau_i = \top$ *or* $\tau_i' = \bot$
- $\tau_o = \tau_o'$ *or* $\tau_o = \bot$ *or* $\tau_o' = \top$
- $\tau_r = \tau_r'$ *or* $\tau_r = \bot$ *or* $\tau_r' = \top$

Lemma 8 (Well-typed initial states). *Let* e *be an expression with* $\vdash_{prog} e : \tau$*, and* $\langle S \mid \mu \rangle = \text{initState}(e)$*. Then* $\Sigma_0^\tau \vdash \langle S \mid \mu \rangle$*.*

Theorem 1 (Preservation). *If* $\Sigma \vdash \langle S \mid \mu \rangle$ *and* $\langle S \mid \mu \rangle \to \langle S' \mid \mu' \rangle$*, then* $\Sigma' \vdash \langle S' \mid \mu' \rangle$ *for some* $\Sigma' \supseteq \Sigma$*.*

Proof. We focus on the main cases (see the extended version for the remainder). *Case distinction* on the evaluation rule.

- *Case* E-Create: So $S = \ell@C[\mathsf{create}\,x.e]\,; S^?$, $S' = \ell@C[\ell^*]\,; S^?$, and $\mu' = \mu \cup \{(\ell^*, v^*)\}$ with $v^* = \lambda x^*.((\lambda x.e\,x^*)\,\ell^*)$. From the assumed $\Sigma \vdash_{\text{le}^*} S$, Lemma 1 yields $\emptyset|\Sigma \vdash \mathsf{create}\,x.e : \tau_c \& \varphi_c$ for some τ_c, φ_c. The only rule to derive this is T-Create, from which we can conclude that $\varphi_c = \bot$ and $\tau_c = \tau_i \rightsquigarrow \tau_o/\tau_r$. Furthermore, the same rule requires that

$$x{:}\tau_c|\Sigma \vdash e : \tau_i \xrightarrow{\varphi} \tau_r \& \varphi' \tag{3}$$

  for some $\varphi, \varphi' \sqsubseteq \tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r$. Choose $\Sigma' = \Sigma \cup \{(\ell^*, \tau_c)\}$, which still is a function due to freshness condition on ℓ^*. Also, the constraints on occurrences

of $\top$ and $\bot$ in $\tau_{i,o,r}$, as demanded in (DefΣ), are satisfied by the precondition in T-Create. Then $\emptyset|\Sigma' \vdash \ell^* : \tau_c \& \bot$ holds, and $\Sigma' \vdash_{\text{le}^*} S'$ follows by Lemma 5. $\Sigma' \vdash_{\text{w}} S'$ follows from $\Sigma \vdash_{\text{w}} S$ (using that $\vdash_{\text{w}}$ is obviously montone in Σ).

$\Sigma' \vdash_{\text{sto}^*} \mu'$ requires that μ' is a function (true due to the freshness of ℓ^*), and that v^* has the right type: $\emptyset|\Sigma' \vdash v^* : \tau_i \xrightarrow{\varphi^*} \tau_r \& \bot$ for some $\varphi^* \sqsubseteq \tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r$. This follows from (3) by T-App and T-Fix, observing that all type derivations using Σ also work with its superset Σ'. ✓

– *Case* E-Yie: Then $S = \ell_1@e_1; \ell_2@e_2; S^?$ and $S' = \ell_2@e_2'; S^?$ with $e_1 = C_1[\mathsf{yield}\, v_y]$, $e_2 = C_2[\mathsf{resume}\, \ell\, v_a\, v_s\, v_n]$, $e_2' = C_2[v_s\, v_y]$. Also $\mu' = \mu \cup \{(\ell_1, e_1')\}$ with $e_1' = \lambda x^*.C_1[x^*]$.

We choose $\Sigma' = \Sigma$. Due to $\Sigma \vdash_{\text{le}^*} S$, Σ must contain entries for ℓ_1, ℓ_2 of the form $\Sigma(\ell_k) = \tau_i^k \rightsquigarrow \tau_o^k/\tau_r^k$ for $k = 1, 2$. Furthermore, by T-StackN, $\emptyset|\Sigma \vdash e_1 : \tau_r^1 \& \varphi_e^1$ and $\emptyset|\Sigma \vdash e_2 : \tau_r^2 \& \varphi_e^2$ must hold for some $\varphi_e^1 \sqsubseteq \varphi^1, \varphi_e^2 \sqsubseteq \varphi^2$ (where $\varphi^k = \tau_i^k \ddot{\rightsquigarrow} \tau_o^k/\tau_r^k$).

To prove $\Sigma' \vdash_{\text{le}^*} S'$ and $\Sigma' \vdash_{\text{w}} S'$ using Lemma 6, we need to show $\emptyset|\Sigma \vdash v_y : \tau_o^1 \& \bot$ (the rest follows immediately from the assumptions and $\Sigma' = \Sigma$). Let τ_y, τ_i^* be the types assigned to v_y and $\mathsf{yield}\, v_y$, respectively, in the type derivation for the assumed $\Sigma \vdash_{\text{le}} \ell_1@e_1$. By T-Yie and Lemma 1, we get

$$\tau_i^* \ddot{\rightsquigarrow} \tau_y/\bot \sqsubseteq \tau_i^1 \ddot{\rightsquigarrow} \tau_o^1/\tau_r^1 \tag{4}$$

By Lemma 7, $\tau_y = \bot$ (impossible: Lemma 2), or $\tau_o^1 = \top$ (contradicting (DefΣ)), or $\tau_y = \tau_o^1$. ✓

To prove $\Sigma \vdash_{\text{sto}^*} \mu'$, it remains to prove that μ' is still a function (by T-State, $\ell_1 \notin \text{dom}\,(\mu)$, so adding ℓ_1 preserves the function property of μ), and that $\emptyset|\Sigma \vdash \mu'(\ell_1) : \tau_i^1 \xrightarrow{\varphi} \tau_r^1 \& \bot$ with some $\varphi \sqsubseteq \varphi_1$. Applying Lemma 7 to (4), we know that $\tau_i^1 = \tau_i^*$ (T-Yie forbids $\tau_i^* = \top$, (DefΣ) forbids $\tau_i^1 = \bot$). Setting $\Gamma := x^* : \tau_i$, we immediately get $\Gamma|\Sigma \vdash x^* \leq \mathsf{yield}\, v_y$, and by Lemma 4, $\Gamma|\Sigma \vdash C_1[x^*] \leq e_1$. Hence, by T-Fix, $\emptyset|\Sigma \vdash \lambda x^*.C_1[x^*] : \tau_i^1 \xrightarrow{\varphi'} \tau_r^1 \& \bot$ for some $\varphi' \sqsubseteq \varphi_1$. ✓

– *Case* E-Res: So $S = \ell_2@C[\mathsf{resume}\, \ell_1\, v_a\, v_s\, v_n]; S^?$, and $S' = \ell_1@(v_1\, v_a); S$ with $v_1 = \mu(\ell_1), \mu' = \mu \setminus \ell_1$. Furthermore, we know that $\Sigma(\ell_1) = \tau_i \rightsquigarrow \tau_o/\tau_r$ for some τ_i, τ_o, τ_r because $\Sigma \vdash_{\text{le}^*} S$ holds.

We choose $\Sigma' = \Sigma$. For $\Sigma' \vdash \langle S' \mid \mu' \rangle$, we need to prove: (a) $\Sigma \vdash_{\text{sto}^*} \mu'$, (b) $\Sigma \vdash_{\text{le}} \ell_1@v_1\, v_a$, (c) $\ell_1 \notin \text{labels}(S^?)$ (which yields $\Sigma' \vdash_{\text{le}^*} S'$ together with (b)), (d) $\Sigma \vdash_{\text{w}} S'$, and (e) $\text{labels}(S') \cap \text{dom}\,(\mu') = \emptyset$.

Proposition (a) follows immediately from μ' being a subset of μ and the assumption $\Sigma \vdash_{\text{sto}^*} \mu$. ✓Proposition (c) is clear from $\Sigma \vdash \langle S \mid \mu \rangle$.✓Proposition (e) is clear because moving ℓ_1 between sets preserves disjointness. ✓

Proposition (b): prove $\emptyset|\Sigma \vdash v_1\, v_a : \tau_r \& \varphi_1$ for some $\varphi_1 \sqsubseteq \tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r$. By assumption $\Sigma \vdash_{\text{sto}^*} \mu$, we know about v_1 that $\emptyset|\Sigma \vdash v_1 : \tau_i \xrightarrow{\varphi_1'} \tau_r \& \bot$ holds with $\varphi_1' \sqsubseteq \tau_i \ddot{\rightsquigarrow} \tau_o/\tau_r$. With Lemma 1, Lemma 2 and T-Res, we conclude that $\emptyset|\Sigma \vdash v_a : \tau_i \& \bot$, which yields the desired result using T-App.

$$
\begin{aligned}
R ::= & k^n\, v_1 \ldots v_n \mid (\mathsf{fix}\, \lambda f.\lambda x.e)\, v \\
& \mid \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \\
& \mid \mathsf{create}\, x.e \mid \mathsf{yield}\, v \mid \mathsf{resume}\, \ell\, v\, v \mid \mathsf{transfer}\, \ell\, v
\end{aligned}
$$

Fig. 9. The language of redexes

Proposition (d): By Lemma 1, Lemma 2 and T-Res, we know that $\emptyset|\Sigma \vdash v_s : \tau_o \xrightarrow{\varphi_s} \tau_q \& \bot$ and $\emptyset|\Sigma \vdash v_n : \tau_r \xrightarrow{\varphi_n} \tau_q \& \bot$, which matches the precondition of T-WaitN about v_s and v_n. The other preconditions follow directly from the assumptions. ✓

End case distinction on the evaluation rule. ■

3.2 Progress

In this section, we state the progress property. First, we define a language of redexes in Fig. 9, then we show in Lemma 10 that well-typed expressions are either values or redexes embedded in evaluation contexts, which facilitates the main progress theorem, Theorem 2.

Lemma 9 (Canonical forms)

1. *If* $\Gamma|\Sigma \vdash v : \tau \xrightarrow{\varphi} \tau' \& \varphi'$, *then* $v = \mathsf{fix}\lambda f.\lambda x.e$ *for some* f, x, e.
2. *If* $\Gamma|\Sigma \vdash v : \mathsf{Bool} \& \varphi'$, *then* $v = \mathsf{true}$ *or* $v = \mathsf{false}$.
3. *If* $\Gamma|\Sigma \vdash v : \mathsf{Unit} \& \varphi'$, *then* $v = \mathsf{unit}$.
4. *If* $\Gamma|\Sigma \vdash v : \tau_i \rightsquigarrow \tau_o/\tau_r \& \varphi'$, *then* $v = \ell$ *for some* $\ell \in \mathrm{dom}(\Sigma)$.

Lemma 10 ($C[R]$-decomposition). *Let* $\emptyset|\Sigma \vdash e : \tau \& \varphi$ *for some* e, Σ, τ, φ. *Then* e *is a value, or* $e = C[R]$ *for some* C, R.

Theorem 2 (Progress). *Let* $\langle S \mid \mu \rangle$ *be an evaluation state and* Σ *a store typing so that* $\Sigma \vdash \langle S \mid \mu \rangle$. *Then* $S = \ell @ v; \epsilon$ *for some* v, ℓ, *or* $\langle S \mid \mu \rangle \to \langle S' \mid \mu' \rangle$ *for some* S', μ', *or* $\langle S \mid \mu \rangle \to \mathsf{Error}$.

4 Related Work

Formalizations of coroutines. De Moura and Ierusalemschy [14] formally define coroutines in an untyped λ-calculus with mutable variables as a model for Lua coroutines. Their interexpressibility results (e.g. transfer in terms of resume/yield) make heavy use of untyped mutable variables; it is yet unclear which of the transformations can be adapted to a statically-typed setting. Their work contains a comprehensive overview of the state of the art in coroutines and related techniques.

Wang and Dahl [18] formalize the control-flow aspects of idealized Simula coroutines. The operational semantics of Belsnes and Østvold [1] also focuses on the control-flow aspects but includes threads and thread-coroutine interaction. Laird [10] presents a process calculus in which the coroutine is the basic building block. Berdine and coworkers [2] define coroutines in their process calculus.

Language design. Languages with parameterless coroutines include Simula [5], Modula-2 [19], and BETA [9]. However, the type systems of these languages need not treat coroutines with much sophistication because the coroutine operations do not pass values.

Some mainstream dynamically-typed languages like Python [17] and Lua [15] pass values to and from coroutines, but without a static type system. C# [13] has static typing and generators (asymmetric coroutines with parameters only for yield), but as the yield-equivalent may only be used lexically inside the generator's body, the type system avoids the complexity involved with stackful coroutines.

Marlin's ACL [12] is a (statically typed) coroutine extension of Pascal in which coroutines can accept parameters. In analogy to the separation between procedures and functions in Pascal, it features separate syntax for symmetric and asymmetric coroutines. The problem of procedures performing coroutine operations on behalf of the enclosing coroutine is solved by referring to the static block structure, which simplifies the type system at the expense of flexibility.

Haynes and coworkers [7] express coroutines using continuations in Scheme; Harper and colleagues [6] in turn describe a type system for continuations.

Lazy languages like Haskell [16] get asymmetric coroutines for free: a coroutine can be viewed as a transformer of a stream of input values to a stream of output values, which is straightforward to implement using lazy lists. Blazevic [3] produced a more sophisticated monad-based implementation of symmetric coroutines.

5 Conclusion

We presented CorDuroy, a language with type-safe stackful asymmetric and symmetric first-class coroutines, and proved its soundness. CorDuroy constitutes the first provably sound type system for an eager-evaluated language that supports realistic and expressive facilities for coroutines.

One obvious direction of further research is the addition of polymorphism. For subtype polymorphism, (a subset of) C# would be a promising candidate since it already has generators. Parametric polymorphism would likely bring challenges similar to those caused by mutable references.

As this work was inspired by De Moura and Ierusalemschy's paper [14] in which they present translations between various styles of coroutines, continuations and threads in an untyped setting with mutable variables, it would be interesting to see if the corresponding typed equivalences also hold.

Currently, the operational semantics contains failure rules. Instead, linearity could be introduced to prevent the activation of returned coroutines by keeping track of the coroutine state.

References

1. Belsnes, D., Østvold, B.M.: Mixing threads and coroutines (2005), submitted to FOSSACS 2005, bjarte@nr.no
2. Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear continuation-passing. Higher-Order and Symbolic Computation 15(2-3), 181–208 (2002)

3. Blazevic, M.: monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations (2010), http://hackage.haskell.org/package/monad-coroutine
4. Conway, M.E.: Design of a separable transition-diagram compiler. Comm. ACM 6(7), 396–408 (1963)
5. Dahl, O.J., Myrhaug, B., Nygaard, K.: SIMULA 67 Common Base Language. Norwegian Computing Center, Oslo (1970) (revised version 1984)
6. Harper, R., Duba, B.F., MacQueen, D.: Typing first-class continuations in ML. In: Proc. 1991 ACM Symp. POPL. ACM Press, Orlando (1991)
7. Haynes, C.T., Friedman, D.P., Wand, M.: Obtaining coroutines with continuations. Computer Languages 11(3), 143–153 (1986)
8. Knuth, D.E.: Fundamental Algorithms, The Art of Computer Programming, 2nd edn., vol. 1. Addison-Wesley, Reading (1968)
9. Kristensen, B.B., Pedersen, B.M., Madsen, O.L., Nygaard, K.: Coroutine sequencing in BETA. In: Proc. of 21st Annual Hawaii International Conference on Software Track, pp. 396–405. IEEE Computer Society Press, Los Alamitos (1988)
10. Laird, J.: A calculus of coroutines. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 882–893. Springer, Heidelberg (2004)
11. Liskov, B.: CLU reference manual. LNCS, vol. 114. Springer, Heidelberg (1981)
12. Marlin, C.D.: Coroutines: a programming methodology, a language design and an implementation. Springer, Heidelberg (1980)
13. Microsoft Corp.: C# Version 2.0 Specification (2005), http://msdn.microsoft.com/en-US/library/618ayhy6(v=VS.80).aspx
14. de Moura, A.L., Ierusalimschy, R.: Revisiting coroutines. ACM Trans. Program. Lang. Syst. 31(2), 1–31 (2009)
15. de Moura, A.L., Rodriguez, N., Ierusalimschy, R.: Coroutines in Lua. Journal of Universal Computer Science 10, 925 (2004)
16. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries, The Revised Report. Cambridge University Press, Cambridge (2003)
17. Van Rossum, G., Eby, P.: PEP 342 – coroutines via enhanced generators (2005), http://www.python.org/dev/peps/pep-0342/
18. Wang, A., Dahl, O.J.: Coroutine sequencing in a block structured environment. BIT Numerical Mathematics 11(4), 425–449 (1971), http://www.springerlink.com/content/g870vkxx22861w50
19. Wirth, N.: Programming in Modula-2. Springer, Heidelberg (1982)
20. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)