Deriving Type Systems and Implementations for Coroutines

Konrad Anton and Peter Thiemann

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

Abstract. Starting from reduction semantics for several styles of coroutines from the literature, we apply Danvy's method to obtain equivalent functional implementations (definitional interpreters) for them. By applying existing type systems for programs with continuations, we obtain sound type systems for coroutines through the translation. The resulting type systems are similar to earlier hand-crafted ones. As a side product, we obtain implementations for these styles of coroutines in OCaml.

1 Introduction

Coroutines are an old programming construct, dating back to the 1960s [5]. They have been neglected for a while, but are currently enjoying a renaissance (e.g. in Lua [12]), sometimes in the limited form of generators (Python [20], C# [17]) and sometimes under different names (e.g., fibers in .NET).

Type systems for coroutines have not been considered in the past. Coroutines without parameters and return values (Simula, Modula 2 [21]), coroutine operations whose effects are tied to the static structure of the program (ACL, [16]), or coroutines lexically limited to the body of one function (C#), could be integrated into said languages without special support in the type system.

In an earlier paper [1], we developed the first type system for a simply-typed λ-calculus with coroutines. This development was done in an ad-hoc style for a feature-rich calculus and resulted in a type and effect system with a simple notion of subeffecting to capture the control effects of coroutine operations.

In this paper, we follow a different course and rigorously derive a type system for a simple core calculus. The derivation starts with a reduction semantics from the literature [11]. To this reduction semantics, we apply Danvy's method [6] to obtain a denotational semantics after several semantics-preserving transformation steps. Then we further transform the semantics to a denotational implementation (a combinator implementation) using methods developed by Thiemann [19]. This combinator implementation of the coroutine operations is directly and practically usable. It is available for OCaml on the web.

The denotational implementation also provides good grounds for constructing a type system that is aware of coroutines. As the combinators contains control operators, we apply a type system for (the control operators) shift and reset [2,7] to them and abstract from the types to obtain the desired system. This approach

$$
\begin{array}{lll}
\text{Expressions} & e & ::= l \mid x \mid \lambda x.e \mid e\, e \mid x := e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid e = e \mid \textbf{nil} \\
\text{Values} & v & ::= l \mid \lambda x.e \mid \textbf{nil} \\
\text{Ev. contexts} & C & ::= \Box \mid e\, C \mid C\, v \mid x := C \mid \textbf{if}\ C\ \textbf{then}\ e\ \textbf{else}\ e \mid e = C \mid C = v
\end{array}
$$

$$
\begin{array}{lll}
(C[x], \theta, l_0) & \Rightarrow & (C[\theta(x)], \theta, l_0) \\
(C[(\lambda x.e)\ v], \theta, l_0) & \Rightarrow & (C[e[x := z]], \theta[z \mapsto v], l_0) \text{ where } z \text{ is fresh} \\
(C[x := v], \theta, l_0) & \Rightarrow & (C[v], \theta[x \mapsto v], l_0) \\
(C[\textbf{if nil then}\ e_2\ \textbf{else}\ e_3], \theta, l_0) & \Rightarrow & (C[e_3], \theta, l_0) \\
(C[\textbf{if}\ v\ \textbf{then}\ e_2\ \textbf{else}\ e_3], \theta, l_0) & \Rightarrow & (C[e_2], \theta, l_0) \text{ if } v \neq \textbf{nil} \\
(C[l = l], \theta, l_0) & \Rightarrow & (C[l], \theta, l_0) \\
(C[l_1 = l_2], \theta, l_0) & \Rightarrow & (C[\textbf{nil}], \theta, l_0) \text{ if } l_1 \neq l_2
\end{array}
$$

Fig. 1. Syntax and reductions of the base calculus.

allows us to construct a variety of type systems. We provide a type soundness proof for one of them by specifying a typed translation to cps augmented with a reader monad. This translation is *not ad-hoc* either, because we fork it off from an intermediate transformation result of Danvy's method.

In summary, the contributions of this paper are:

- Systematically derived implementation of coroutine library for OCaml.
- Systematically derived monomorphic type system for coroutines.
- Type soundness proof via a typed translation to a standard monomorphic lambda calculus with references.

2 Reduction Semantics for Coroutines

In their paper "Revisiting Coroutines" (henceforth abbreviated RC), Moura and Ierusalimschy [11] define a core calculus that can be extended to cover various styles and features of coroutines. They consider two styles of coroutines, symmetric and asymmetric. In the symmetric setting, a coroutine passes control by explicitly invoking another coroutine. In the asymmetric setting, there are two ways of passing control. Either a coroutine can explicitly invoke another coroutine, in which case it establishes a parent-child relationship to the next coroutine, or it can pass control to the implicit parent coroutine. In each case, the coroutine that passes control suspends itself. As a final variation, not considered in RC, a coroutine implementation may support both styles of control passing [1, 14].

The base calculus is a call-by-value lambda calculus with assignment, locations, equality, and a conditional. Evaluation proceeds from right to left. Fig. 1 defines its syntax and its reduction rules. The latter work on a configuration that consists of an expression e, a store $\theta \in (\mathit{Var} \cup \mathit{Loc}) \rightarrow \mathit{Value}$ that maps variables and locations to values, and a location $l \in \mathit{Loc}$. Here, Var is a set of variables and Loc is a set of store locations. The base calculus does not use the third location component, but the symmetric coroutine operations do.

$$
\begin{array}{ll}
\text{Expressions } e & ::= \cdots \mid \textbf{create}\ e \mid \textbf{transfer}\ e\ e \mid \textbf{current} \\
\text{Ev. contexts } C & ::= \cdots \mid \textbf{create}\ C \mid \textbf{transfer}\ e\ C \mid \textbf{transfer}\ C\ v
\end{array}
$$

$$
\begin{array}{lcl}
(C[\textbf{create}\ v], \theta, l_0) & \Rightarrow & (C[l], \theta[l \mapsto v], l_0) \text{ where } l \notin \mathrm{dom}(\theta) \\
(C[\textbf{transfer}\ l\ v], \theta, l_0) & \Rightarrow & (\theta(l)\ v, \theta[l \mapsto \textbf{nil}, l_0 \mapsto \lambda x.C[x]], l) \text{ where } l \neq l_0 \\
(C[\textbf{transfer}\ l_0\ v], \theta, l_0) & \Rightarrow & (C[v], \theta, l_0) \\
(C[\textbf{current}], \theta, l_0) & \Rightarrow & (C[l_0], \theta, l_0)
\end{array}
$$

Fig. 2. Syntax and reductions of the calculus with symmetric coroutines.

The calculus models beta reduction in Scheme style by renaming the bound variable to a fresh variable z and assigning the substituted value to that variable. Thus, these fresh variables serve as locations and facilitate a straightforward implementation of the assignment operation.

2.1 Symmetric Coroutines

The symmetric coroutine calculus extends the base calculus with operations to create a coroutine and to transfer control to a coroutine. Moreover, there is an operation to obtain the identity of the currently running coroutine.

Fig. 2 defines the extended syntax as well as the additional evaluation contexts and reduction rules. For simplicity, it differs from the calculus in RC in that it does not have the notion of a main coroutine, to which execution falls back if a coroutine just terminates without passing on control. The equivalence proof of symmetric and asymmetric coroutines in RC relies on this feature.

This calculus (as well as the following ones) models coroutines exclusively as storable values in the sense of the EOPL book [13]. Thus, an expression never denotes a coroutine directly, but only via a store location.

The definition of the **transfer** operation implements the convention that an active coroutine is not represented in the store. Creating a coroutine obtains an unused store location and assigns it a procedure. Transferring control to a coroutine sets its location to **nil** and suspending the coroutine overwrites it with a new procedure (the continuation that arises as the context of the **transfer**). The second reduction rule for **transfer** implements the special case that a coroutine transfers control to itself.

2.2 Asymmetric Coroutines

The asymmetric calculus is slightly more complicated. To the base calculus, it adds operations to create a coroutine, to resume another coroutine (establishing the parent-child relationship mentioned at the beginning), and to yield to the parent coroutine.

Fig. 3 shows the extended syntax, evaluation contexts, and reductions. Deviating from the presentation in RC, the configuration keeps the current coroutine l_0 in the third component and the evaluation context has been split in an inner evaluation context C and an outer evaluation context D. The D contexts

$$\begin{array}{lll}
\text{Expressions} & e & ::= \cdots \mid \textbf{create}\ e \mid \textbf{resume}\ e\ e \mid \textbf{yield}\ e \mid l:\ e \\
\text{Ev. contexts} & C & ::= \cdots \mid \textbf{create}\ C \mid \textbf{resume}\ e\ C \mid \textbf{resume}\ C\ v \mid \textbf{yield}\ C \\
\text{Ev. contexts II} & D & ::= \square \mid C[l:\ D]
\end{array}$$

$$\begin{array}{lll}
(D[C[\textbf{create}\ v]], \theta, l_0) & \Rightarrow & (D[C[l]], \theta[l \mapsto v], l_0)\ \text{where}\ l \notin \text{dom}(\theta) \\
(D[C[\textbf{resume}\ l\ v]], \theta, l_0) & \Rightarrow & (D[C[l_0 :\ \theta(l)\ v]], \theta[l \mapsto \textbf{nil}], l)\ \text{where}\ l \neq l_0 \\
(D[C[\textbf{resume}\ l_0\ v]], \theta, l_0) & \Rightarrow & (D[C[v]], \theta, l_0) \\
(D[C[l :\ C_0[\textbf{yield}\ v]]], \theta, l_0) & \Rightarrow & (D[C[v]], \theta[l_0 \mapsto \lambda x.C_0[x]], l) \\
(D[C[l :\ v]], \theta, l_0) & \Rightarrow & (D[C[v]], \theta, l)
\end{array}$$

Fig. 3. Syntax and reductions of the calculus with asymmetric coroutines.

$$\begin{array}{lll}
\text{Expressions} & e & ::= \cdots \mid \textbf{create}\ e \mid \textbf{transfer}\ e\ e \mid \textbf{resume}\ e\ e \mid \textbf{yield}\ e \mid l:\ e \\
\text{Ev. contexts} & C & ::= \cdots \mid \textbf{create}\ C \mid \textbf{transfer}\ e\ C \mid \textbf{transfer}\ C\ v \\
 & & \quad \mid \textbf{resume}\ e\ C \mid \textbf{resume}\ C\ v \mid \textbf{yield}\ C \\
\text{Ev. contexts II} & D & ::= \square \mid C[l:\ D]
\end{array}$$

$$\begin{array}{lll}
(D[C[\textbf{create}\ v]], \theta, l_0) & \Rightarrow & (D[C[l]], \theta[l \mapsto v], l_0)\ \text{where}\ l \notin \text{dom}(\theta) \\
(D[C[\textbf{transfer}\ l\ v]], \theta, l_0) & \Rightarrow & (D[\theta(l)\ v], \theta[l \mapsto \textbf{nil}, l_0 \mapsto \lambda x.C[x]], l)\ \text{if}\ l \neq l_0 \\
(D[C[\textbf{transfer}\ l_0\ v]], \theta, l_0) & \Rightarrow & (D[C[v]], \theta, l_0) \\
(D[C[\textbf{resume}\ l\ v]], \theta, l_0) & \Rightarrow & (D[C[l_0 :\ \theta(l)\ v]], \theta[l \mapsto \textbf{nil}], l)\ \text{where}\ l \neq l_0 \\
(D[C[\textbf{resume}\ l_0\ v]], \theta, l_0) & \Rightarrow & (D[C[v]], \theta, l_0) \\
(D[C[l :\ C_0[\textbf{yield}\ v]]], \theta, l_0) & \Rightarrow & (D[C[v]], \theta[l_0 \mapsto \lambda x.C_0[x]], l) \\
(D[C[l :\ v]], \theta, l_0) & \Rightarrow & (D[C[v]], \theta, l)
\end{array}$$

Fig. 4. Syntax and reductions of the calculus with Dahl-Hoare style coroutines.

structure the evaluation contexts in pieces between the newly introduced labeled expressions $l : \ldots$. The latter denote return points for the **yield** operation and they keep and restore the identity l of the parent coroutine if a coroutine terminates without yielding. The reductions imported from the base calculus are all lifted to work in context $D[C[\ldots]]$ instead of just plain $C[\ldots]$.

In this calculus, a coroutine may terminate sensibly. If it finishes with a value, it implicitly yields to its parent or, at the top-level, it concludes the computation. There is again a special case for a coroutine to resume to itself.

2.3 Dahl-Hoare Style Coroutines

Dahl-Hoare style coroutines combine the features of the symmetric and the asymmetric calculus as suggested by Haynes and coworkers [14]. Fig. 4 defines syntax and semantics of the corresponding calculus. It extends the asymmetric calculus with the straightforward adaptation of the **transfer** rules to nested evaluation contexts of the form $D[C[\ldots]]$, but still capturing only the innermost C-continuation.

3 From Reduction Semantics to Denotational Implementation

In a series of papers, Danvy developed a systematic method to interconvert different styles of semantic artifacts while preserving their meaning. Here, we are interested in the route from reduction semantics to a definitional interpreter as spelt out in Danvy's invited presentation at ICFP 2008 [6]. We follow that route exactly for the three reduction semantics from Sec. 2 to obtain three equivalent definitional interpreters.

In each case, the sequence of semantic artifacts starts with an ML program that implements the respective reduction semantics. The first step converts the reduction semantics to a small-step abstract machine by applying refocusing [10]. The result is fused with an iteration function to obtain a tail-recursive evaluation function. Inlining the reduction function and then applying transition compression (function unrolling on known arguments and simplification) results in a tail-recursive interpreter that still manipulates a syntactic representation of the evaluation context. The next step is to refunctionalize this evaluation context to a continuation resulting in an interpreter with continuations [9]. The interpreter can be converted to direct style and subjected to closure unconversion to obtain a "natural looking" interpreter that represents values no longer syntactically.

As the intermediate steps are amply demonstrated in the work of Danvy and coworkers (e.g., [6]), we refrain from going through them in detail. We only comment on special steps that need to be taken for the calculi with the asymmetric coroutine operators and show the essential parts of the final results, in particular omitting the standard lambda calculus parts.

3.1 Symmetric Coroutines

The direct-style version of the interpreter in Fig. 5 is equivalent to the reduction semantics in Fig. 2 because it has been constructed from the latter using a sequence of semantics-preserving transformations. As a final step, we have transformed the store and the currently executed coroutine into two global variables, after observing that they are passed in a single-threaded way.

Most of the code should be self-explanatory. The operations `shift` and `push_prompt` (and `new_prompt`) are from Kiselyov's implementation [15] of the control operators shift and reset [8] that arise from during the transformation to direct style. The code avoids special handling of the case where a coroutine transfers control to itself by choosing the right ordering for the reads and writes of the store.

Close scrutiny of the code (it's in fact easier to see when studying the intermediate results of the transformation) shows that `push_prompt` is only called when the evaluation context is empty. Thus, `push_prompt` need only be placed once at the top-level and `shift` could be replaced by `call/cc` because it would be an error if the call to `cor v2` ever returned. Applying this transformation yields in principle the implementation of coroutines given by Haynes, Friedman,

```
(* values manipulated by interpreter *)
type value =
  | VLoc of location
  | VFun of (value -> value)
  | VNil

let pp : value prompt = new_prompt ()

let rec evaldu_expr e =
  match e with
  | ...
  | Create e ->
      let v = evaldu_expr e in
      let newl = fresh_location () in
      update_Loc newl v;
      VLoc newl
  | Transfer (e1, e2) ->
      let v2 = evaldu_expr e2 in
      let VLoc l1 = evaldu_expr e1 in
      let l = !current_coroutine in
      shift pp (fun ec ->
        update_Loc l (VFun ec);
        let VFun cor = lookup_Loc l1 in
        update_Loc l1 VNil;
        current_coroutine := l1;
        push_prompt pp (fun () ->
          cor v2))
  | Current ->
      let l = !current_coroutine in
      VLoc l
```

Fig. 5. Definitional interpreter with symmetric coroutines (excerpt).

```
type value = (* as before *)

let pp : value prompt = new_prompt ()

let rec evaldg_expr e =
  match e with
  | ...
  | Create e -> (* as before *)
  | Resume (e1, e2) ->
      let v2 = evaldg_expr e2 in
      let VLoc l1 = evaldg_expr e1 in
      let VFun cor = lookup_Loc l1 in
      update_Loc l1 VNil;
      let lc = !current_coroutine in
      let v =
        push_prompt pp
          (fun () ->
            current_coroutine := l1;
            cor v2)
      in current_coroutine := lc;
      v
  | Yield e ->
      let v = evaldg_expr e in
      let lc = !current_coroutine in
      shift pp (fun ec ->
        update_Loc lc (VFun ec);
        v)
```

Fig. 6. Definitional interpreter with asymmetric coroutines (excerpt).

and Wand [14]. Their implementation looks more complicated at first glance because they represent a coroutine by a function and because they abstract the coroutine body over the **transfer** function. But disentangling their implementation of **create** and **transfer** leads to the code in Fig. 5 with `call/cc` in place of `shift`.

3.2 Asymmetric Coroutines

The calculus with asymmetric coroutines requires some extra transformation steps. These extra steps are caused by the second level of evaluation contexts named D in Fig. 3. After the initial transformation steps, the intermediate result is a tail-recursive interpreter with two arguments that hold evaluation contexts, corresponding to the $D[C[\dots]]$ in the reduction semantics. Both of them are then refunctionalized, giving rise to an interpreter with two levels of continuations, and then transformed to direct style two times to obtain the code in Fig. 6. As before, the direct-style transformation introduces the control operators `shift` and `reset`.

The interpreter reuses the same type of values. It merges the two cases for **resume** by choosing the correct ordering for the reads and writes to the store. The The `shift` operation in **yield** abstracts the context up to the prompt set in the parent coroutine's **resume** instruction. The variable `lc` in the code for **resume** implements the labeled expression $lc, \cdots :$ and the assignment `current_coroutine := lc` implements its reduction.

3.3 Dahl-Hoare Style Coroutines

To obtain a definitional interpreter with Dahl-Hoare style coroutines, it is sufficient to merge the code from Fig. 5 and Fig. 6. Thanks to the shared configuration, the code fits together without change.

3.4 Correctness

The chain of transformations according to Danvy's recipe preserves the semantics in the following sense. The proof is by appeal to the correctness of each transformation step.

Lemma 1. *Let e be a closed expression and $\overset{*}{\Rightarrow}$ be the reflexive transitive closure of $\Rightarrow$. Suppose that $(e, \theta_0, l_0) \overset{*}{\Rightarrow} (v, \theta, l)$.*

Then `eval_expr` *e with store $\lceil \theta_0 \rceil$ and* `!current_coroutine =` *l_0 evaluates to $\lceil v \rceil$ with store $\lceil \theta \rceil$ and* `!current_coroutine =` *l with $\lceil \cdot \rceil$ defined by*

$$\begin{aligned} \lceil l \rceil &= \texttt{VLoc}\ l \\ \lceil \lambda x.e \rceil &= \texttt{VFun}\ (\texttt{fun x -> eval\_expr}\ e) \\ \lceil \mathbf{nil} \rceil &= \texttt{VNil} \end{aligned}$$

4 Implementation

In earlier work, we exhibited and proved correct a systematic transformation from an interpreter to a denotational implementation [19]. A denotational implementation specifies the semantics of each single language construct in terms of a combinator. Although the referenced work is posed in the area of program generation, the underlying technique is more generally applicable. In particular, it is applicable to the definitional interpreters constructed in Sec. 3.

As we already have closure un-converted interpreters, it remains to transform binders to a higher-order abstract syntax representation and to extract the combinators. Fig. 7 shows the combinators extracted from Fig. 5 and Fig. 6. Fortunately, our implementation language OCaml also performs right-to-left evaluation of function arguments so that it matches the theory perfectly.

The final transformation step in our previous work [19] is tag removal, which gets rid of the `value` type as also advertised by Carette and coworkers [4]. While this step is not essential for deriving a type system, we apply it to obtain a type-safe implementation of coroutines for OCaml. The appendix Sec. A shows

```
let create v =
  let newl = fresh_location () in
  update_Loc newl v;
  VLoc newl
let resume (VLoc l1, v2) =
  let VFun cor = lookup_Loc l1 in
  update_Loc l1 VNil;
  let lc = !current_coroutine in
  let v =
    push_prompt pp
      (fun () ->
        current_coroutine := l1;
        cor v2)
  in current_coroutine := lc;
  v
let yield v =
  let lc = !current_coroutine in
  shift pp (fun ec ->
    update_Loc lc (VFun ec);
    v)
let transfer (VLoc l1, v2) =
  shift pp (fun ec ->
    let lc = !current_coroutine in
    let VFun cor = lookup_Loc l1 in
    update_Loc l1 VNil;
    update_Loc lc (VFun ec);
    current_coroutine := l1;
    push_prompt pp (fun () -> cor v2))
```

Fig. 7. Combinators extracted from the definitional interpreters.

the code, which is also available on the web.[1] Space does not permit further discussion of the implementation. Let it suffice to say that the interface is monadic and matches the type system explained in the next Sec. 5.

5 Deriving a Type System

The development in Sec. 3 and Sec. 4 shows that coroutines are tightly connected to composable continuations by exhibiting a formally derived implementation of the former with the latter. The results of the RC paper also strongly support this point of view.

To obtain a type system for coroutines, we therefore choose the following strategy. We regard the coroutine operations as abbreviations for the combinators in Fig. 7. As these combinators contain the control operators shift and reset, we employ a type system that supports shift and reset to obtain adequate typings for them. Such a system has been proposed by Danvy and Filinski [7] and later scrutinized and extended with polymorphism by Asai and Kameyama [2]. We use that system to informally derive a type system for coroutines and postpone a formal treatment to Sec. 6.

[1] `http://proglang.informatik.uni-freiburg.de/projects/coroutines/`

The Danvy/Filinsky type system proves judgments of the form $\Gamma; \alpha \vdash e : \tau; \beta$ where the type environment Γ, the expression e, and the resulting type τ are as in a standard type system (e.g., the system of simple types). Additionally, α and β stand for the answer type of the implicit continuation. More precisely, evaluation of the expression e modifies the answer type from α to β. The formal explanation is that the type of e^* is $(\tau^* \to \alpha^*) \to \beta^*$, where * indicates application of the call-by-value CPS transformation to a term or a type. The function type in this system also includes the modification of the answer type of the implicit continuation in its body. We write $\sigma/\alpha \to \tau/\beta$ for a function from σ to τ that modifies the answer type from α to β. For reference, Sec. B in the appendix contains the typing rules of the monomorphic system in a variant adapted to the right-to-left evaluation order of our calculus.

5.1 Global Variables and the Reader Monad

Another important observation concerns the global variable that contains the current coroutine. Inspection of the interpreter (Fig. 5 and Fig. 6) reveals that the location of the current coroutine does not change while the body of a fixed coroutine executes. For example, the code for **resume** sets the current coroutine to the resumed routine and restores the current coroutine to its previous value when the resumed coroutine terminates or yields. The **transfer** operation overwrites the current coroutine to the called coroutine without remembering the past one.

This observation shows that the current coroutine is essentially stored in a reader monad. Thus, it need not be threaded through the computation, but just passed downwards. Resuming or transferring to another coroutine starts in a freshly initialized reader monad.

A similar observation applies to the global coroutine store θ. Each location of this store either contains a function/continuation or the value **nil**. The value **nil** indicates that the coroutine is either active or suspended by a **resume** operation. With this convention, the semantics enforces that there is only one instance of a coroutine at any time: it is a run-time error to resume or transfer to a coroutine which is currently **nil**.

Analyzing the interpreter shows that the set of coroutine locations that are set to **nil** does not change while the body of a fixed coroutine executes, as long as the **transfer** operation is not used.[2] The code for **resume** overwrites the location of the resumed coroutine with **nil** and the corresponding **yield** operation overwrites this location with a captured continuation.

Thus, the information which coroutine must not be invoked (because its location is **nil**) could be stored in a reader monad, but not the actual function or continuation which has to be threaded through the computation.

[2] With **transfer** we could build code that performs the following control transfers. Let l_1, l_2, and l_3 be coroutine locations. At the top-level, first resume to l_1, which in turn resumes l_2. Then transfer to l_3, which yields (to l_1), which in turn transfers to l_2.

When l_2 is first active, l_1 and l_2 are both **nil**. The second time round, only l_2 is **nil**.

$$\sigma, \tau, \alpha, \beta ::= \cdots \mid \sigma \xrightarrow{\alpha} \tau \mid \mathrm{cor}(\beta)$$

$$\frac{\Gamma \vdash e : \sigma \xrightarrow{\sigma} \tau \,\&\, \alpha}{\Gamma \vdash \textbf{create}\ e : \mathrm{cor}(\sigma) \,\&\, \alpha} \qquad \frac{\Gamma \vdash e_1 : \mathrm{cor}(\beta) \,\&\, \alpha \qquad \Gamma \vdash e_2 : \beta \,\&\, \alpha}{\Gamma \vdash \textbf{transfer}\ e_1\ e_2 : \alpha \,\&\, \alpha}$$

$$\Gamma \vdash \textbf{current} : \mathrm{cor}(\alpha) \,\&\, \alpha \qquad \frac{\Gamma, x : \sigma \vdash e : \tau \,\&\, \beta}{\Gamma \vdash \lambda x.e : \sigma \xrightarrow{\beta} \tau \,\&\, \alpha}$$

$$\frac{\Gamma \vdash e_1 : \sigma \xrightarrow{\alpha} \tau \,\&\, \alpha \qquad \Gamma \vdash e_2 : \sigma \,\&\, \alpha}{\Gamma \vdash e_1\ e_2 : \tau \,\&\, \alpha}$$

Fig. 8. Types and relevant typing rules for symmetric coroutines.

In the case of (pure) symmetric coroutines, this information does not matter. There is always exactly one active coroutine, the location of which is set to **nil**. According to the semantics, a program can transfer safely to any coroutine without crashing.

5.2 Symmetric Coroutines

If we apply the Danvy/Filinsky type system to the **transfer** operator in Fig. 7, we obtain the following results.

- The type of a coroutine is a reference to a function type.
- The function type of a coroutine always has the form $\tau/\alpha \to \alpha/\alpha$.
- Under the assumption that `current_coroutine` has type $\mathrm{ref}(\tau_1/\alpha_1 \to \alpha_1/\alpha_1)$, the type of **transfer** is $\mathrm{ref}(\tau_0/\alpha_0 \to \alpha_0/\alpha_0) \times \tau_0/\alpha_1 \to \tau_1/\beta_1$.
- The return type of the coroutine type as well as its answer types do not matter because a symmetric coroutine never returns.

These findings motivate the abbreviation $\mathrm{cor}(\tau) = \mathrm{ref}(\forall\alpha.\tau/\alpha \to \alpha/\alpha)$ for the type of a coroutine that accepts a value of type τ. In addition, the type system needs to keep track of the type of the current coroutine, which happens to be stored in a reader monad (Sec. 5.1). This observation leads to an indexed monadic typing judgment of the form $\Gamma \vdash e : \tau \,\&\, \alpha$ where the effect α indicates that the current coroutine has type $\mathrm{cor}(\alpha)$ and function types of the corresponding form $\sigma \xrightarrow{\alpha} \tau$ where α is the expected type of the current coroutine at the call site of the function. Fig. 8 contains the type syntax and the relevant typing rules.

5.3 Asymmetric Coroutines

Again, applying the Danvy/Filinsky type system to the **resume** and **yield** operators in Fig. 7, we obtain the following results.

- The type of a coroutine is a reference to a function type.

$$\sigma, \tau, \alpha, \beta, \gamma, \delta ::= \cdots \mid \sigma \xrightarrow{\alpha \rightsquigarrow \beta} \tau \mid \mathrm{cor}(\alpha, \beta)$$

$$\frac{\Gamma \vdash e : \beta \xrightarrow{\beta \rightsquigarrow \gamma} \gamma \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma \vdash \textbf{create}\ e : \mathrm{cor}(\beta, \gamma) \mathbin{\&} \alpha \rightsquigarrow \delta} \qquad \Gamma \vdash \textbf{current} : \mathrm{cor}(\alpha, \delta) \mathbin{\&} \alpha \rightsquigarrow \delta$$

$$\frac{\Gamma \vdash e_1 : \mathrm{cor}(\beta, \gamma) \mathbin{\&} \alpha \rightsquigarrow \delta \qquad \Gamma \vdash e_2 : \beta \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma \vdash \textbf{resume}\ e_1\ e_2 : \gamma \mathbin{\&} \alpha \rightsquigarrow \delta} \qquad \frac{\Gamma \vdash e : \delta \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma \vdash \textbf{yield}\ e : \alpha \mathbin{\&} \alpha \rightsquigarrow \delta}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \mathbin{\&} \beta \rightsquigarrow \gamma}{\Gamma \vdash \lambda x.e : \sigma \xrightarrow{\beta \rightsquigarrow \gamma} \tau \mathbin{\&} \alpha \rightsquigarrow \delta} \qquad \frac{\Gamma \vdash e_1 : \sigma \xrightarrow{\alpha \rightsquigarrow \delta} \tau \mathbin{\&} \alpha \rightsquigarrow \delta \qquad \Gamma \vdash e_2 : \sigma \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma \vdash e_1\ e_2 : \tau \mathbin{\&} \alpha \rightsquigarrow \delta}$$

Fig. 9. Types and relevant typing rules for asymmetric coroutines.

$$\frac{\Gamma \vdash e_1 : \mathrm{cor}(\beta, \delta) \mathbin{\&} \alpha \rightsquigarrow \delta \qquad \Gamma \vdash e_2 : \beta \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma \vdash \textbf{transfer}\ e_1\ e_2 : \alpha \mathbin{\&} \alpha \rightsquigarrow \delta}$$

Fig. 10. Typing rule for transfer.

- The function type of a coroutine always has the form $\tau/\alpha \to \alpha/\alpha$.
- **resume** has type $\mathrm{ref}(\sigma/\tau \to \tau/\tau) \times \sigma/\alpha \to \tau_0/\beta$.
- Assuming that the current coroutine has type $\mathrm{ref}(\sigma/\tau \to \tau/\tau)$, the type of **yield** is $\tau_0/\gamma \to \sigma/\tau_0$.
- If the coroutine also returns normally, then $\tau_0 = \tau$, in which case the types work out to
  - **resume** : $\mathrm{ref}(\sigma/\tau \to \tau/\tau) \times \sigma/\alpha \to \tau/\beta$ and
  - **yield** : $\tau/\gamma \to \sigma/\tau$ (assuming the coroutine was resumed by the above **resume**).

A suitable coroutine type for this constellation is $\mathrm{cor}(\sigma, \tau) = \mathrm{ref}(\sigma/\tau \to \tau/\tau)$ and the corresponding type system is again a monadic system that keeps track of the type of the current coroutine stored in the reader monad. The typing judgment correspondingly reads $\Gamma \vdash e : \tau \mathbin{\&} \alpha \rightsquigarrow \beta$ where the effect $\alpha \rightsquigarrow \beta$ specifies that the current coroutine has type $\mathrm{cor}(\alpha, \beta)$. The function type has the form $\sigma \xrightarrow{\alpha \rightsquigarrow \beta} \tau$ where the current coroutine at the point of the function call is expected to be $\mathrm{cor}(\alpha, \beta)$. Fig. 9 shows the relevant parts of the syntax of types and of the typing rules.

5.4 Dahl-Hoare Style Coroutines

The Dahl-Hoare style only adds the **transfer** operation to the API for asymmetric coroutines. Thus, it remains to find a typing rule for **transfer** to go along with the system in Sec. 5.3. The rule (shown in Fig. 10) is similar to the typing of the **resume** operation, but —as it replaces the current coroutine— its return/yield type must be equal to the return type of the current coroutine.

$$\sigma, \tau, \alpha, \beta, \gamma, \delta ::= \cdots \mid \sigma \overset{L,l,\alpha \rightsquigarrow \beta}{\longrightarrow} \tau \mid \mathrm{cor}^l_L(\alpha, \beta)$$

$$\frac{\Gamma; L', l' \vdash e : \beta \overset{L,l,\beta \rightsquigarrow \gamma}{\longrightarrow} \gamma \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma; L', l' \vdash \mathbf{create}^l\ e : \mathrm{cor}^l_L(\beta, \gamma) \mathbin{\&} \alpha \rightsquigarrow \delta} \qquad \Gamma; L, l \vdash \mathbf{current} : \mathrm{cor}^l_L(\alpha, \delta) \mathbin{\&} \alpha \rightsquigarrow \delta$$

$$\frac{\begin{array}{c}\Gamma; L', l' \vdash e_1 : \mathrm{cor}^l_L(\beta, \gamma) \mathbin{\&} \alpha \rightsquigarrow \delta \\ \Gamma; L', l' \vdash e_2 : \beta \mathbin{\&} \alpha \rightsquigarrow \delta \qquad l \notin L' \qquad L = L' \cup \{l'\}\end{array}}{\Gamma; L', l' \vdash \mathbf{resume}^l\ e_1\ e_2 : \gamma \mathbin{\&} \alpha \rightsquigarrow \delta} \qquad \frac{\Gamma; L, l \vdash e : \delta \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma; L, l \vdash \mathbf{yield}\ e : \alpha \mathbin{\&} \alpha \rightsquigarrow \delta}$$

$$\frac{\Gamma; L, l' \vdash e_1 : \mathrm{cor}^l_L(\beta, \delta) \mathbin{\&} \alpha \rightsquigarrow \delta \qquad \Gamma; L, l' \vdash e_2 : \beta \mathbin{\&} \alpha \rightsquigarrow \delta \qquad l \notin L \setminus \{l'\}}{\Gamma; L, l' \vdash \mathbf{transfer}^l\ e_1\ e_2 : \alpha \mathbin{\&} \alpha \rightsquigarrow \delta}$$

$$\frac{\Gamma, x : \sigma; L, l \vdash e : \tau \mathbin{\&} \beta \rightsquigarrow \gamma}{\Gamma; L', l' \vdash \lambda x.e : \sigma \overset{L,l,\beta \rightsquigarrow \gamma}{\longrightarrow} \tau \mathbin{\&} \alpha \rightsquigarrow \delta}$$

$$\frac{\Gamma; L, l \vdash e_1 : \sigma \overset{L,l,\alpha \rightsquigarrow \delta}{\longrightarrow} \tau \mathbin{\&} \alpha \rightsquigarrow \delta \qquad \Gamma; L, l \vdash e_2 : \sigma \mathbin{\&} \alpha \rightsquigarrow \delta}{\Gamma; L, l \vdash e_1\ e_2 : \tau \mathbin{\&} \alpha \rightsquigarrow \delta}$$

Fig. 11. Type system for asymmetric coroutines with **nil** tracking.

5.5 Keeping Track of Nil

Up to this point, the type systems do not prevent to resume or transfer to a pending coroutine that waits for a **yield** and has its location set to **nil**. As mentioned in Sec. 5.1, this information could be split from the coroutine store and passed in a reader monad. Reflecting that reader monad in the type system requires a number of changes.

1. There must be a static approximation of the location where the coroutine is stored. We solve that by attaching a source label to each **create** expression and using that label.
2. There must be an approximation of the set of locations of pending coroutines. A set of labels is sufficient.
3. The typing judgment must keep track of the additional indexing of the reader monad.
4. The function type and the coroutine type must be extended to accommodate the additional indexing information.

Fig. 11 contains a first draft of a type system that tracks this extra information. The typing judgment extends to $\Gamma; L, l \vdash e : \sigma \mathbin{\&} \alpha \rightsquigarrow \delta$ where L is the set of pending labels and $l \in L$ is the label of the currently active coroutine. The function type and the coroutine type carry the same information L, l as indexes. Thus, a coroutine of type $\mathrm{cor}^l_L(\beta, \gamma)$ is stored in location l and, while active, the coroutines in L are pending. A function of type $\sigma \overset{L,l,\alpha \rightsquigarrow \delta}{\longrightarrow} \tau$ can only be called while in coroutine l with pending set L.

$$\frac{L \supseteq L' \qquad l \notin L'}{\mathrm{cor}^l_L(\alpha,\beta) \leq \mathrm{cor}^l_{L'}(\alpha,\beta)}$$

$$\frac{\sigma' \leq \sigma \qquad \tau \leq \tau' \qquad L \supseteq L' \qquad l \notin L'}{\sigma \xrightarrow{L,l,\alpha \rightsquigarrow \beta} \tau \leq \sigma' \xrightarrow{L',l,\alpha \rightsquigarrow \beta} \tau'}$$

Fig. 12. Subtyping.

The **create** operation transforms a function into a coroutine while preserving its indexes. The **resume** operation checks with $l \notin L'$ that the resumed coroutine is neither active nor in the pending set and demands a suitable pending set $L = L' \cup \{l'\}$ for the resumed coroutine. The **transfer** operation checks similarly that the target coroutine is not in the pending set, but transferring to oneself is permitted. The remaining rules are straightforward.

This type system has been constructed systematically from the operational semantics, but it turns out to be quite conservative. First, it disallows a coroutine to resume to itself, which is fine by the operational semantics. However, this restriction is needed to obtain a sound type system.

Suppose we changed the **resume** rule to allow self-resumption. In this case, the constraints in the rule would change to $l \notin L' \setminus \{l'\}$ and $L = L' \setminus \{l'\} \cup \{l\}$. In a program that creates more than one coroutine instance with the same label l, resuming the first of these coroutines and then resuming to coroutine with a different label blocks later resumes to another l-coroutine, which is annoying, but still sound. However, because a coroutine is allowed to resume to itself, the modified type system lets a first instance with label l directly resume to another instance with label l. This instance, in turn could try to resume to the first instance, which is not stopped by the type system but which results in a run-time error. Hence, the system that allows self-resumption would be unsound.

This complication is caused by the procedure-call-like semantics of **resume**. It is not an issue for the **transfer** operation because it replaces the currently running coroutine.

Second, the type system requires subtyping to avoid being overly rigid. Without subtyping, each **resume** operation can only invoke one kind of coroutine, namely the one indexed with the correct l and L. The amendment is to allow multiple **create** operations labeled with the same label and to introduce subtyping with respect to the L index. Fig. 12 contains suitable subtyping rules for the function type and for the coroutine type. When moving to the supertype, both rules admit decreasing the index: a function or coroutine can always be used in a less restrictive context. The function type is contravariant in the argument and covariant in the result, as usual. The argument and result types of the coroutine need to remain invariant for technical reasons (stored in a reference).

$$
\begin{aligned}
(\Gamma \vdash e : \tau \,\&\, \alpha \rightsquigarrow \beta)^* &= \Gamma^* \vdash e^* : \mathrm{ref}(\alpha^* \to \beta^*) \to (\tau^* \to \beta^*) \to \beta^* \\
(x_1 : \sigma_1, \ldots, x_n : \sigma_n)^* &= x_1 : \sigma_1^*, \ldots, x_n : \sigma_n^* \\
(\mathrm{cor}(\alpha/\beta))^* &= \mathrm{ref}(\alpha^* \to \beta^*) \\
(\sigma \xrightarrow{\alpha \rightsquigarrow \beta} \tau)^* &= \sigma^* \to \mathrm{ref}(\alpha^* \to \beta^*) \to (\tau^* \to \beta^*) \to \beta^* \\
l^* &= \lambda c.\lambda k.k\, l \\
x^* &= \lambda c.\lambda k.k\, x \\
(\lambda x.e)^* &= \lambda c.\lambda k.k\, (\lambda x.e^*) \\
(e_1\; e_2)^* &= \lambda c.\lambda k.e_2^*\, c\, (\lambda v_2.e_1^*\, c\, (\lambda v_1.v_1\; v_2\; c\, k)) \\
(\textbf{create}\; e_1)^* &= \lambda c.\lambda k.e_1^*\, c(\lambda v_1.\textbf{let}\, l = \mathit{fresh}\, \textbf{in}\, \mathit{update}\; l\, (\lambda v_2.v_1\; v_2\; l\, (\lambda z.z)); k\, l) \\
(\textbf{resume}\; e_1\; e_2)^* &= \lambda c.\lambda k.e_2^*\, c\, (\lambda v_2.e_1^*\, c\, (\lambda v_1.\textbf{let}\, v = \mathit{lookup}\, v_1\, \textbf{in}\, k\, (v\; v_2))) \\
(\textbf{yield}\; e_1)^* &= \lambda c.\lambda k.e_1^*\, c(\lambda v_1.\mathit{update}\; c\, k; v_1) \\
(\textbf{transfer}\; e_1\; e_2)^* &= \lambda c.\lambda k.e_2^*\, c\, (\lambda v_2.e_1^*\, c\, (\lambda v_1.\textbf{let}\, v = \mathit{lookup}\, v_1\, \textbf{in}\, \mathit{update}\; c\, k; v\, v_2))
\end{aligned}
$$

Fig. 13. Translation.

6 Type Soundness

To prove type soundness of the system in Sec. 5.3 and Sec. 5.4 (Fig. 9 and Fig. 10), we consider an intermediate product of the transformation chain from Sec. 3. In the case of asymmetric coroutines, an intermediate product after refunctionalization was an interpreter with two levels of continuations. According to the observation in Sec. 5.1, we transformed the use of the state monad for the current coroutine in this interpreter into a use of the reader monad. Subsequently, we removed the outer layer of continuations by direct style transformation and moved the interpretation of the lambda from the cases for **resume** and **transfer** into the case for **create**. The consequence of the last transformation step is that a coroutine location *always* contains a continuation, before it could also contain a standard (CPS) function.

With this preparation, we consider the interpreter as a translation from the source language to CPS plus reader monad in a lambda calculus with references and read off the accompanying translation on types. Fig. 13 shows the details of the translation. For the typing to go through, the type of a computation must be polymorphic over the final answer type of the continuation.

Lemma 2. *Suppose that $\Gamma \vdash e : \tau \,\&\, \alpha \rightsquigarrow \beta$ in the system of Fig. 9 with the* **transfer** *rule. Then $\Gamma^* \vdash e^* : \mathit{ref}(\alpha^* \rightsquigarrow \beta^*) \to (\tau^* \to \beta^*) \to \beta^*$ in a simple type system for call-by-value lambda calculus with references (e.g., [18, Chapter 13]).*

The type soundness of the **nil**-tracking type system in Fig. 11 and Fig. 12 can be shown using a similar translation.

7 Related Work

Language design aspects of coroutines have been explored in the 1970s and 1980s. Coroutines have found entry in some current programming languages (Python [20], Lua [12], C# [17]), but their formal semantics has been neglected.

The exception is the RC paper [11], which rigorously defines small step operational semantics for several styles of coroutines and proves various expressivity results among them and with respect to continuations and subcontinuations. One might also view the Scheme implementation of Haynes and coworkers [14] a formal specification of coroutines.

However, apart from our own work [1], we are not aware of any exploration of type systems tailored to coroutines. The other paper considers a richer calculus inspired by the needs of practical programming and develops its type system in an ad-hoc way. For example, the **resume** operation takes two continuations to distinguish between a yield and a normal return of the invoked coroutine. On the other hand, the system developed in the present paper reveals that the effect in the previous system [1] keeps track of the type of the current continuation which is stored in a reader monad. This insight would not have been possible without the systematic transformation approach.

Blazevic [3] produced a monad-based implementation of symmetric coroutines with session types in Haskell. Our work is based on an eager language, offers asymmetric coroutines, and is derived from a specification.

8 Conclusion

Using the systematic transformation approach advocated by Danvy, we have transformed a small-step reduction semantics for various styles of coroutines to a working, type-safe OCaml implementation. We have further derived a type system for a calculus with coroutines by applying a type system that is aware of control operators to the result of the transformation. Another outcome of the transformation is the translation which is used in constructing a type soundness proof for the type system.

We found that the systematic approach enabled additional insights. An example is the discovery of the use of the reader monad, which leads to the construction of the **nil**-tracking type system. Also the type soundness proof is vastly simplified with the translation that is also derived from an intermediate transformation step. Last but not least, the transformation gave rise to a practically useful, type-safe library implementation.

References

1. K. Anton and P. Thiemann. Typing coroutines. In R. Page, V. Zsók, and Z. Horváth, editors, *Eleventh Symposium on Trends in Functional Programming (draft proceedings)*, pages 91–105. University of Oklahoma Printing Services, 2010.

2. K. Asai and Y. Kameyama. Polymorphic delimited continuations. In Z. Shao, editor, *APLAS*, volume 4807 of *LNCS*, pages 239–254, Singapore, 2007. Springer.
3. M. Blazevic. monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations. `http://hackage.haskell.org/package/monad-coroutine`, 2010.
4. J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.
5. M. E. Conway. Design of a separable transition-diagram compiler. *Comm. ACM*, 6(7):396–408, 1963.
6. O. Danvy. Defunctionalized interpreters for programming languages. pages 131–142, 2008.
7. O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.
8. O. Danvy and A. Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, 1990. ACM Press.
9. O. Danvy and K. Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009.
10. O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Nov. 2004.
11. A. L. de Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):1–31, 2009.
12. A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10:925, 2004.
13. D. P. Friedman and M. Wand. *Essentials of Programming Languages.* MIT Press and McGraw-Hill, 3rd edition, 2008.
14. C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *ACM Conference on Lisp and Functional Programming*, pages 293–298, 1984.
15. O. Kiselyov. Delimited control in OCaml, abstractly and concretely: System description. In M. Blume, N. Kobayashi, and G. Vidal, editors, *FLOPS*, volume 6009 of *LNCS*, pages 304–320, Sendai, Japan, Apr. 2010. Springer.
16. C. D. Marlin. *Coroutines: a programming methodology, a language design and an implementation.* Springer, 1980.
17. Microsoft Corp. C# Version 2.0 Specification, 2005. `http://msdn.microsoft.com/en-US/library/618ayhy6(v=VS.80).aspx`.
18. B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.
19. P. Thiemann. Combinators for program generation. *J. Funct. Program.*, 9(5):483–525, Sept. 1999.
20. G. Van Rossum and P. Eby. PEP 342 – coroutines via enhanced generators. `http://www.python.org/dev/peps/pep-0342/`, 2005.
21. N. Wirth. *Programming in Modula-2.* Springer, 1982.

SUPPLEMENTARY MATERIAL NOT PART OF THE SUBMISSION

A Coroutine Implementation for OCaml

A.1 Interface

```
type ('i,'o,'a) cm  (* coroutine monad *)
type ('i,'o) coro   (* coroutine representation *)

(* monadic glue operators *)
val return : 'a -> ('i,'o,'a) cm
val (>>=) : ('i,'o,'a) cm -> ('a -> ('i,'o,'b) cm) -> ('i,'o,'b) cm

(* coroutine operators *)
val create : ('i -> ('i,'o,'o) cm) -> ('i,'o) coro
val current : ('i,'o, ('i,'o) coro) cm
val resume: ('i,'o) coro -> 'i -> 'o
val yield : 'o -> ('i,'o,'i) cm
val transfer : ('ii,'o) coro -> 'ii -> ('i,'o,'i) cm
```

A.2 Implementation

```
(* pure OCaml coroutines using Oleg Kiselyov's delimited continuations *)
open Delimcc

type ('i, 'o) coro = (('i->'o)ref * 'o prompt)
type ('i,'o,'a) cm = (('i,'o) coro) -> ('a)

let create (f: 'i -> ('i,'o,'o) cm) : ('i,'o) coro =
  let p: 'o prompt = new_prompt() in
  let rec coro = ( ref (fun (x:'i) -> (f x) coro ), p)
  in coro

let current (curcoro : ('i,'o) coro) : ('i,'o) coro =
  curcoro

let resume (coro: ('ii,'oo) coro) (v: 'ii)  : 'oo =
  let (pfun, pr) = coro in
  let new_f = !pfun in
    push_prompt pr (fun () ->
                      new_f v)

let yield (v:'o) (state: ('i,'o) coro) : 'i =
  let (pfun, pr) = state in
    shift pr (fun (k:('i->'o)) ->
                pfun := k ;
                v)

let transfer (coro: ('ii,'o) coro) (v: 'ii) (state: ('i,'o) coro) : 'i =
  let (pfun, my_pr) = state in
    shift my_pr (fun (k: 'i->'o) ->
                   pfun := k ;
                   let (otherpfun, other_pr) = coro in
                   let otherfun = !otherpfun in
                     push_prompt other_pr (fun () ->
                                             otherfun v) )

let return (v : 'a) (state : ('i,'o) coro) = v

let (>>=) (m : ('i,'o,'a) cm) (f : 'a -> ('i,'o,'b) cm) : ('i,'o,'b) cm =
  fun (state: ('i,'o) coro) ->
    (f (m state)) state
```

B Type System with Shift and Reset

$$\Gamma, x : \tau \vdash_p x : \tau \qquad\qquad \frac{\Gamma, x : \sigma; \alpha \vdash e : \tau; \beta}{\Gamma \vdash_p \lambda x.e : (\sigma/\alpha \to \tau/\beta)}$$

$$\frac{\Gamma; \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta); \delta \qquad \Gamma; \beta \vdash e_2 : \sigma; \gamma}{\Gamma; \alpha \vdash e_1\, e_2 : \tau; \delta} \qquad\qquad \frac{\Gamma \vdash_p e : \tau}{\Gamma; \alpha \vdash e : \tau; \alpha}$$

$$\frac{\Gamma, k : (\tau/\delta \to \alpha/\delta); \sigma \vdash e : \sigma; \beta}{\Gamma; \alpha \vdash \texttt{shift}\ k.e : \tau; \beta} \qquad\qquad \frac{\Gamma; \sigma \vdash e : \sigma; \tau}{\Gamma \vdash_p \texttt{reset}\ e : \tau}$$

Fig. 14. Type system with shift and reset.

Fig. 14 contains the typing rules of Danvy and Filinski's type system with shift and reset [7]. The $\vdash_p$ judgment is for expressions that never modify the answer type: values and applications of the `reset` operator.

As the soundness of the type system for shift and reset depends on the left-to-right evaluation order of the language, we had to modify the function application rule to cater for the right-to-left evaluation of our calculus. Our replacement rule is

$$\frac{\Gamma; \beta \vdash e_1 : (\sigma/\alpha \to \tau/\beta); \gamma \qquad \Gamma; \gamma \vdash e_2 : \sigma; \delta}{\Gamma; \alpha \vdash e_1\, e_2 : \tau; \delta}$$

We have proved the correctness of this rule by applying the right-to-left call-by-value CPS translation and type checking the resulting term.