Principles of Callcc

Paulo Emílio de Vilhena[0000−0001−7379−310X]

University of Surrey, Guildford, United Kingdom p.devilhena@surrey.ac.uk

Abstract. The programming construct `callcc` has been the target of multiple criticisms and, unfortunately, of misunderstanding. Contrary to a common misconception that `callcc` is incompatible with the *bind rule*, we show that `callcc` admits strikingly simple Hoare-style reasoning principles in a separation logic that includes the bind rule and a restricted *frame rule*. We exercise this logic in case studies including the specification and verification of an implementation of *control inversion* using `callcc`.

Keywords: Continuations · Separation logic · `callcc`.

1 Introduction

We recall the **semantics** of and suggest a **programming intuition** for `callcc`. Then, we explain **what is known** about its theory and **what is misunderstood**.

Semantics. The semantics of `callcc` is governed by the following reduction rule:[1]

$$K[\texttt{callcc (fn k => } e\texttt{)}] \;\to\; K[e\{\texttt{kont } K/\texttt{k}\}] \qquad \text{(CallccStep)}$$

According to this rule, `callcc (fn k =>` e`)` captures the surrounding *evaluation context* K (which identifies the next subexpression to be evaluated, thus defining the order in which subexpressions of a program are evaluated) as a *continuation* (the first-class value `kont` K) and runs e with `k` bound to `kont` K.

A continuation `k` thus simply represents a captured evaluation context K. The expression `throw (kont` K`)` x can be used to restore K by discarding the current context and inserting the value x where `callcc` initially appeared:

$$K'[\texttt{throw (kont } K\texttt{) } x] \;\to\; K[x] \qquad \text{(ThrowStep)}$$

Programming intuition. These reduction rules can be interpreted according to a programming intuition aptly evoked by Leroy's and Madore's slogans:

> `callcc` *is the* `goto` *of lambda calculus.* – Leroy [17]
>
> `callcc` *is a dynamic* `goto`. – Madore [19]

[1] Programs studied in this paper have access to a heap, so, technically, the reduction rules should specify the action of programs on this heap. For programs that (at least from a formal-semantics perspective) do not modify the heap, we simply omit this specification. This is the case for `callcc` and `throw`. ↩

These slogans endorse the view that, when representing a program in the form $K[e]$, that is, when representing a program as a pair of an evaluation context K and an expression e about to be run, we can see K as the *position* or *program point* of e. From this perspective, the operation `callcc (fn k => e)` can be understood as saving a checkpoint that records e's position, to which e will then have access via the binder `k`. The operation `throw k x` restores this checkpoint by placing the value x where e originally appeared, as if e had returned x.

The interpretation of `callcc` as a form of `goto` can be concretely realised as the following interface implemented in Standard ML of New Jersey (SML/NJ) [2], a version of ML with support for `callcc` (via the library `SMLofNJ.Cont`[2]):

```
signature DYNGOTO = sig
  type 'a t
  exception E
  val new_lbl    : unit -> 'a t
  val checkpoint : 'a t -> (unit -> 'a) -> 'a
  val goto       : 'a t -> 'a -> 'b
end;

structure Dyngoto :> DYNGOTO = struct
  type 'a t = 'a cont option ref
  exception E
  fun new_lbl () = ref NONE
  fun checkpoint lbl e = callcc (fn k => (lbl := SOME k; e ()))
  fun goto lbl x =
    case !lbl of SOME k => throw k x | NONE => raise E
end;
```

The structure `Dyngoto` internally uses `callcc` to implement *labels*. Labels record program points. They are introduced with `new_lbl` in an uninitialised state. The operation `checkpoint lbl e` records in `lbl` the program point where the operation is performed, and forces the execution of the thunk `e`. With `goto lbl x`, control flows back to the point where the last `checkpoint` of `lbl` was performed, using the value `x` as if it were the result of this `checkpoint` operation itself.

The type variable `'a` with which the type of labels `Dyngoto.t` is parameterised records the result type of the thunk in `checkpoint`. Clearly, only values of this type can be used when going back to `checkpoint` via `goto`, so the type of `x` must match the type parameter of `lbl` in `goto`. Because `goto` does not return, it can be used in any context; that is, it is polymorphic in its return type `'b`.

Here is an illustrative implementation of factorial using `Dyngoto`:

```
fun fact n =
  let val lbl = new_lbl ()
      val (acc, i) = checkpoint lbl (fn _ => (1, n))
  in if i > 1 then goto lbl (acc * i, i - 1) else acc end;
```

[2] The structure `SMLofNJ.Cont` [25] introduces, among other components, the functions `callcc : ('a cont -> 'a) -> 'a` and `throw : 'a cont -> 'a -> 'b`, where `'a cont` is the type of continuations that can be resumed with `'a` values. ↩

$$\text{(Callcc)}\ \frac{\{P\}\,K[e\{\texttt{kont}\ K/\texttt{k}\}]\,\{\Phi\}}{\{P\}\,K[\texttt{callcc (fn k => } e\texttt{)}]\,\{\Phi\}} \qquad \text{(Throw)}\ \frac{\{P\}\,K[x]\,\{\Phi\}}{\{P\}\,K'[\texttt{throw (kont } K\texttt{) } x]\,\{\Phi\}}$$

$$\text{(Return)}\ \frac{\Box(P \mathrel{-\!\!*} \Phi\,x)}{\{P\}\,x\,\{\Phi\}} \qquad \text{(Bind)}\ \frac{\{P\}\,e_1\,\{\Phi\} \qquad \forall x.\ \{\Phi\,x\}\,(e_2\{x/\texttt{x}\})\,\{\Psi\}}{\{P\}\,\texttt{let val x = } e_1\ \texttt{in}\ e_2\ \texttt{end}\,\{\Psi\}}$$

$$\text{(App)}\ \frac{\{P\}\,(e\{x/\texttt{x}\})\,\{\Phi\}}{\{P\}\,\texttt{(fn x => } e\texttt{)}\,x\,\{\Phi\}} \qquad \text{(Move)}\ \frac{\mathit{Persistent}\,(R) \qquad \Box(R \mathrel{-\!\!*} \{P\}\,e\,\{\Phi\})}{\{P * R\}\,e\,\{\Phi\}}$$

$$\text{(Alloc)}\ \frac{}{\{\mathit{True}\}\,\texttt{ref}\ x\,\{r.\,r \mapsto x\}} \qquad \text{(Read)}\ \frac{}{\{r \mapsto x\}\,\texttt{!}r\,\{y.\,r \mapsto x * y = x\}} \qquad \text{(Write)}\ \frac{}{\{r \mapsto x\}\,r\ \texttt{:=}\ y\,\{\_.\,r \mapsto y\}}$$

$$\text{(Persistent)}\ \frac{}{\mathit{Persistent}\,(\{P\}\,e\,\{\Phi\})}$$

Fig. 1. Unsound set of rules due to a counterexample by Timany and Birkedal [27].

To compute the factorial of `n`, `fact` iterates `i` from `n` to `1`, accumulating in `acc` the product of every value of `i`. It uses `checkpoint` to record the point where `acc` and `i` are declared and uses `goto` to repeatedly go back to that point, updating `acc` with `acc * i` and `i` with `i - 1` until `i = 1`, when it returns `acc`.

What is known. The `goto` statement has been heavily criticised [7]. Given that `callcc` can be viewed as an even more disorderly version of `goto`, it is not surprising that `callcc` has been the target of equally severe criticisms [16]. From a theoretical perspective, however, and, specifically, from the perspective of *program logics*, what principles are lost due to `callcc`?

Timany and Birkedal [27] provide an insightful analysis of the reasoning principles of `callcc` in the context of *separation logic* [21,24] (specifically *Iris* [15]). They show in particular that the set of rules in Figure 1 is unsound.[3] Each rule can be read as an assertion (where every open variable is universally quantified) stating that the iterated separating conjunction of the premises entails the conclusion. These rules are mostly standard with the exception of Rules (Callcc) and (Throw), which respectively paraphrase the reduction rules of `callcc` and `throw`, Rules (CalleeStep) and (ThrowStep), in terms of Hoare triples.[4] Timany

[3] The rules are stated here using SML/NJ for consistency. Timany and Birkedal [27] use a formal calculus with similar syntax and semantics. ↩

[4] Rules (Return), (Move), and (Persistent) are slightly less standard because they reveal the distinction between exclusive and *persistent* assertions. Assertions in separation logic can claim ownership over exclusive resources. Therefore, in Iris, by

and Birkedal show that, with the rules from Figure 1, it is possible to derive a specification that is *false*; that is, it is possible to verify a program according to a specification that disagrees with the actual behaviour of this program. Therefore, by contradiction, this set of rules must be unsound.

This is the counterexample by Timany and Birkedal written here in SML/NJ:

```
callincr ≜ let val r = ref 0
               val f = callcc (fn k => fn _ => throw k (fn _ => ()))
           in (r := !r + 2; f (); !r) end
```

From the reduction rules of `callcc` and `throw`, it is clear that *callincr* returns `4`, because the assignment to `r` happens twice. Indeed, when `f ()` is executed, control flows back to where `f` is declared, binding `f` to `fn _ => ()`, so that the reference `r` is updated exactly twice.

With the reasoning rules from Figure 1, Timany and Birkedal show that it is possible to derive the following incorrect specification of *callincr*:

$$\{\mathit{True}\}\ \mathit{callincr}\ \{y.\, y = 2\} \tag{IncorrectSpec}$$

The derivation of (IncorrectSpec) starts with the application of (Bind) to reason about the binding of `r`. The proof is thus split into two parts:

$$\{\mathit{True}\}\ \texttt{ref 0}\ \{r.\, r \mapsto 0\}$$

$$\{r \mapsto 0\} \begin{pmatrix} \texttt{let val f = callcc (fn k => fn \_ =>} \\ \texttt{\quad\quad\quad\quad throw k (fn \_ => ()))} \\ \texttt{in (}r\ \texttt{:= !}r\ \texttt{+ 2; f (); !}r\texttt{) end} \end{pmatrix} \{y.\, y = 2\}$$

The first goal, specifying `ref 0`, is an immediate consequence of (Alloc). The second goal is further split into two parts, again via (Bind):

$$\{r \mapsto 0\} \begin{pmatrix} \texttt{callcc (fn k => fn \_ =>} \\ \texttt{\quad throw k (fn \_ => ()))} \end{pmatrix} \{f.\, r \mapsto 0 * \{r \mapsto 2\}\, f\ ()\, \{\_.\, r \mapsto 2\}\}$$

$$\{r \mapsto 0 * \{r \mapsto 2\}\, f\ ()\, \{\_.\, r \mapsto 2\}\}\ (r\ \texttt{:= !}r\ \texttt{+ 2;}\ f\ \texttt{();}\ \texttt{!}r)\ \{y.\, y = 2\}$$

The second goal, which specifies $(r\ \texttt{:= !}r\ \texttt{+ 2;}\ f\ \texttt{();}\ \texttt{!}r)$, follows immediately from the rules for reading and writing references, Rules (Read) and (Write), and from the admitted specification of f.[5] We are left with the first goal, which

default, assertions follow an affine-usage policy, which prevents their duplication; that is, P does not entail $P * P$. Persistent assertions avoid this restriction: they can be arbitrarily shared and, in particular, duplicated. They are defined in Iris using the *persistently modality* $\square$: an assertion P is persistent if P entails $\square P$. The assertion $\square P$ indicates P does not claim ownership over exclusive resources: a derivation of $\square P$ can rely only on other persistent assertions. ↩

[5] To step through the instructions in a sequence expression, we apply (Bind), noting that $(e_1\texttt{;}\ e_2)$ and $\texttt{let val \_ =}\ e_1\ \texttt{in}\ e_2\ \texttt{end}$ have the same behaviour. To move the specification of f out of the triple, we apply (Move) and (Persistent). ↩

states a triple for the `callcc` expression. This goal follows mechanically from Rules (Callcc), (Throw), (Return), and (App). Indeed, because the `callcc` expression appears here in isolation from the rest of the program, the application of (Callcc) substitutes `k` with a continuation `kont` $\bullet$ that captures the empty context $\bullet$. The application of (Throw) then simply replaces the `throw` expression with `fn _ => ()` (the result of filling the empty context with this value), thus simplifying the triple to a trivially derivable one.

The key idea in the derivation of (IncorrectSpec) is to exploit a conflict between Rule (Bind) and Rules (Callcc) and (Throw). Whereas (Callcc) and (Throw) rely on a global view of program execution that reveals the surrounding evaluation context K, (Bind) enables a local view, by allowing one to focus on the subexpressions e_1 and e_2 of `let val x =` e_1 `in` e_2 `end`. This conflict is exploited when reasoning about the binding of `f` via (Bind). Indeed, after focusing on the `callcc` expression, we lose track of the global evaluation context `let val f = ` $\bullet$ `in (`r `:= !`r `+ 2; f (); !`r`) end`. When (Callcc) is applied, the empty context $\bullet$ is incorrectly captured instead.

What is misunderstood. Perhaps because of the prominent role played by the bind rule – Rule (Bind) – in the derivation of (IncorrectSpec), authors conclude, for example, that *"in the presence of [*`callcc`*] [. . .] the so-called 'bind rule' [. . .] is no longer valid"* [26]. When taken out of context, this reading seems to imply that, to avoid the contradictory derivation of (IncorrectSpec), program logics for `callcc`, or for languages with similar control operators for *undelimited continuations*, should not admit the bind rule (even though Timany and Birkedal [27] do not make such a claim). In fact, from a logical perspective, all rules from Figure 1 are equally blameable for this contradiction. While Timany and Birkedal [27] show the exclusion of (Bind) leads to a sound logic, there might be a way to provide sound reasoning principles for `callcc` without rejecting the bind rule.

We show such an alternative exists; that it is possible to keep (Bind).

The bind rule is admissible in the presence of `callcc`.

The key idea is to replace Rules (Callcc) and (Throw) with more abstract rules that do not paraphrase the operational behaviour of `callcc` and `throw`. This is not an original contribution from this paper. Before Timany and Birkedal [27], Delbianco and Nanevski [6], for example, had already developed a logic where reasoning rules for `callcc` and the bind rule coexist. The contributions of this paper are (1) to clarify that the inadmissibility of the bind rule in the presence of `callcc` is a misconception; (2) to reveal original reasoning principles for `callcc` and `throw` in an Iris-based separation logic that not only is compatible with the bind rule but also, by virtue of being built on top of Iris, supports the verification of programs with *heap-stored continuations*, a pattern that is common when programming with `callcc` and whose support, for this reason, is key to the verification of programs with `callcc`; and (3) to illustrate how these strikingly simple principles lead to elegant correctness proofs of seemingly challenging programs.

$$\text{(Callcc)}\quad \frac{\forall k.\ \{P * \mathit{isCont}\, k\, \Phi\}\ (e\{k/\mathtt{k}\})\ \{\Phi\}}{\{P\}\ \mathtt{callcc}\ (\mathtt{fn}\ \mathtt{k}\ \mathtt{=>}\ e)\ \{\Phi\}} \qquad \text{(Throw)}\quad \frac{}{\{\mathit{isCont}\, k\, \Phi * \Phi\, x\}\ \mathtt{throw}\ k\ x\ \{\_.\ \mathit{False}\}}$$

$$\text{(Return)}\quad \frac{\Box(P \mathrel{-\!\!*} \Phi\, x)}{\{P\}\ x\ \{\Phi\}} \qquad \text{(Bind)}\quad \frac{\{P\}\ e_1\ \{\Phi\} \qquad \forall x.\ \{\Phi\, x\}\ (e_2\{x/\mathtt{x}\})\ \{\Psi\}}{\{P\}\ \mathtt{let\ val\ x\ =}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end}\ \{\Psi\}}$$

$$\text{(Consequence)}\quad \frac{\Box(P \mathrel{-\!\!*} Q) \qquad \{Q\}\ e\ \{\Psi\} \qquad \Box \forall y.\ \Psi\, y \mathrel{-\!\!*} \Phi\, y}{\{P\}\ e\ \{\Phi\}} \qquad \text{(App)}\quad \frac{\{P\}\ (e\{x/\mathtt{x}\})\ \{\Phi\}}{\{P\}\ (\mathtt{fn}\ \mathtt{x}\ \mathtt{=>}\ e)\ x\ \{\Phi\}}$$

$$\text{(Move)}\quad \frac{\mathit{Persistent}\,(R) \qquad \Box(R \mathrel{-\!\!*} \{P\}\ e\ \{\Phi\})}{\{P * R\}\ e\ \{\Phi\}} \qquad \text{(Frame)}\quad \frac{[P]\ e\ [\Phi]}{[P * R]\ e\ [y.\ \Phi\, y * R]} \qquad \text{(Square)}\quad \frac{[P]\ e\ [\Phi]}{\{P\}\ e\ \{\Phi\}}$$

$$\text{(Alloc)}\quad \frac{}{\{\mathit{True}\}\ \mathtt{ref}\ x\ \{r.\ r \mapsto x\}} \qquad \text{(Read)}\quad \frac{}{\{r \mapsto x\}\ \mathtt{!}r\ \{y.\ r \mapsto x * y = x\}} \qquad \text{(Write)}\quad \frac{}{\{r \mapsto x\}\ r\ \mathtt{:=}\ y\ \{\_.\ r \mapsto y\}}$$

$$\text{(Persistent)}\quad \mathit{Persistent}\,(\{P\}\ e\ \{\Phi\})$$

Fig. 2. Sound set of reasoning principles for `callcc` and `throw`.

$$\mathit{Persistent}\,(\mathit{isCont}\, k\, \Phi) \qquad \frac{\mathit{isCont}\, k\, \Phi \qquad \Box \forall y.\Psi\, y \mathrel{-\!\!*} \Phi\, y}{\mathit{isCont}\, k\, \Psi}$$

Fig. 3. Properties of *isCont*.

2 Principles

Figure 2 shows sound Hoare-style reasoning principles for `callcc` and `throw` in a separation logic that includes the bind rule.

Rule (Callcc) posits that to verify a `callcc` expression, it suffices to verify its inner subexpression e with `k` bound to a universally quantified value k whose only information is $\mathit{isCont}\, k\, \Phi$, which asserts that k is a continuation captured by a `callcc` operation with postcondition Φ. Recall that `throw` k x goes back to where this `callcc` operation was performed, replacing the `callcc` expression with x. It is thus natural that $\Phi\, x$ be included as a precondition in Rule (Throw), because x works as the return value of `callcc`. Moreover, because a `throw` expression itself never returns, the rule postulates that the postcondition of `throw` is *False*. Figure 3 shows that, for any k and Φ, the assertion $\mathit{isCont}\, k\, \Phi$ is persistent and that Φ can be replaced with Ψ provided Ψ implies Φ.

Rules (Callcc) and (Throw) capture the intuition that a continuation k represents the program point of a `callcc` expression, a point to which control can jump with a value x provided x satisfies the postcondition Φ associated to k via $\mathit{isCont}\, k\, \Phi$. They show that a program point is entirely specified by the postcondition Φ of the `callcc` expression. This insight explains a difference between Rule (Callcc) from Figure 1 and Rule (Callcc) from Figure 2. Whereas (Callcc) from Figure 1 relies on a global view of program execution that reveals the current evaluation context K, Rule (Callcc) from Figure 2 targets a `callcc` expression in isolation from K. This is sound precisely because K can be abstracted by the postcondition Φ. The same principle appears in Rule (Bind), which allows a subexpression e_1 to be verified in isolation from the context `let val x = •` $\ldots$ `in` e_2 `end` provided the user can find an adequate postcondition Φ that works as a logical boundary between e_1 and this context.

The parallel between evaluation contexts and postconditions can also be witnessed when comparing the values to which the binder `k` in a `callcc` expression is bound depending on which rule is used. Whereas, with (Callcc) from Figure 1, the binder `k` is bound to the captured context `kont` K, with (Callcc) from Figure 2, it is bound to an abstract value k specified by Φ via the assertion $\mathit{isCont}\, k\, \Phi$.

Let us now explain the remaining rules of Figure 2, except for Rules (Return), (Bind), (App), (Alloc), (Read), (Write), (Move), and (Persistent), which are the same as in Figure 1.

Rule (Consequence), when read from bottom to top, allows the precondition to be weakened and the postcondition to be strengthened.

Rules (Frame) and (Square) add support for a restricted form of the frame rule. They are stated in terms of *square triples* $[P]\, e\, [\Phi]$, which admit the same rules as the usual triples $\{P\}\, e\, \{\Phi\}$ except for Rules (Callcc) and (Throw). Therefore, Rule (Frame) implicitly disallows occurrences of `callcc` and `throw` expressions in e. Rule (Square) allows subexpressions e without `callcc` and `throw` of a larger program (which itself might contain `callcc` and `throw`) to be verified using square triples and, thereby, using the frame rule.

Why not include the frame rule? Adding the frame rule to Figure 2 is unsound:

$$\text{(UnsoundFrame)}\ \frac{\{P\}\, e\, \{\Phi\}}{\{P * R\}\, e\, \{y.\, \Phi\, y * R\}}$$

Indeed, with Rule (UnsoundFrame), (IncorrectSpec) is derivable. With the rules from Figure 2, it is possible to derive the following specification:

$$\{\mathit{True}\}\ \left(\begin{array}{l}\texttt{callcc (fn k => fn \_ =>}\\ \quad\texttt{throw k (fn \_ => ()))}\end{array}\right)\ \{f.\, \{\mathit{True}\}\, f\ \texttt{()}\ \{\_.\, \mathit{True}\}\}$$

Then, with Rules (UnsoundFrame) and (Consequence), it is possible to derive the following specification, which, as explained in § 1, can then be used to derive (IncorrectSpec):

$$\{r \mapsto \texttt{0}\}\ \left(\begin{array}{l}\texttt{callcc (fn k => fn \_ =>}\\ \quad\texttt{throw k (fn \_ => ()))}\end{array}\right)\ \{f.\, r \mapsto \texttt{0} * \{r \mapsto \texttt{2}\}\, f\ \texttt{()}\ \{\_.\, r \mapsto \texttt{2}\}\}$$

```
datatype 'a node = Nil | Cons of 'a * 'a stream
withtype 'a stream = unit -> 'a node;

signature INVERSION = sig
  val invert : (('a -> unit) -> unit) -> 'a stream
end;
```

Fig. 4. Interface for control inversion.

```
datatype 'a tree = Leaf of 'a | Branch of 'a tree * 'a tree;

fun iter f t =
  case t of Leaf x => f x | Branch (l, r) => (iter f l; iter f r);

fun equal s1 s2 =
  case (s1, s2) of (Nil, Nil) => true
   |(Cons (a, s1'), Cons (b, s2')) => a = b andalso equal s1' s2'
   |(_, _) => false;

fun same_fringe t1 t2 =
  equal (invert (fn f => iter f t1)) (invert (fn f => iter f t2));
```

Fig. 5. Application of control inversion to solve the Same Fringe Problem.

3 Case Studies

We illustrate the reasoning principles from Figure 2 in a number of interesting case studies, including the specification and verification of an implementation of *control inversion* using `callcc` and of the `Dyngoto` library from § 1.

3.1 Control Inversion

Control inversion is a programming technique to transform an *iteration method* into a *lazy sequence*. An iteration method is a higher-order function that eagerly applies its argument function to the elements of an underlying collection. Examples include the terms `fn f => List.app f` *xs* in Standard ML [11] and `fun f => List.iter f` *ys* in OCaml [18] for any pair of lists *xs* and *ys*. A lazy sequence allows elements to be consumed one at a time, that is, on demand. Examples include lists in Haskell [12] and values of type `Seq.t` in OCaml [5].

Interface Figure 4 introduces the signature `INVERSION`, which offers an interface for control inversion through the function `invert`. As specified by its type, `invert` transforms an iteration method – a value of type `('a -> unit) -> unit` allowing a function `'a -> unit` to be iterated over a collection of elements of type `'a` – into a lazy sequence – a value of type `'a stream`. A lazy sequence is thus represented as a thunk that, when forced, produces either (1) a `Cons` pair containing the head element of the sequence and a new thunk representing the

```
structure Inversion : INVERSION = struct
  fun invert iter =
    let val r = ref NONE
        fun yield x = callcc (fn kp =>
          throw (valOf (!r)) (Cons (x, fn _ =>
            callcc (fn kc => (r := SOME kc; throw kp ()))
          ))
        )
    in
      fn _ => callcc (fn kc =>
        (r := SOME kc; iter yield; throw (valOf (!r)) Nil)
      )
    end
end;
```

Fig. 6. Implementation of control inversion.

tail elements of the sequence or (2) a `Nil` value indicating the sequence has been exhausted.

Application Figure 5 shows the application of control inversion to derive a concise solution to the *Same Fringe Problem* [13], which asks whether two binary trees *"have exactly the same leaves reading from left to right"* [10]. The solution reduces the problem to that of writing an iteration method for a binary tree, a routine task in any functional programming language. Given the iteration method for trees `iter`, the function `same_fringe` applies `invert` to convert the input trees to lazy sequences. Then, it directly compares these sequences. The function `same_fringe` terminates as soon as a mismatch is found. Moreover, it avoids the introduction of costly intermediate structures to store the elements of each tree.

Implementation Figure 6 introduces an implementation of control inversion using `callcc`. The implementation can be understood in terms of two communicating agents: *consumer* and *producer*. The consumer is the code that calls `invert` and that forces the produced `'a stream` thunks expecting a `'a node` value in return. It is thus responsible for consuming the `'a` elements of the collection to which `iter` provides access. The producer, on the other hand, is the code that gets executed whenever the consumer forces a `'a stream` thunk. It is therefore responsible for producing the `'a node` values expected by the consumer.

The implementation of `invert` uses `callcc` and `throw` to bounce between producer and consumer. It uses the reference `r` to store the consumer's position `kc`, to which it gains access using `callcc` whenever the consumer forces a thunk. Moreover, as soon as an element x becomes accessible via the iteration method `iter`, the implementation again uses `callcc` to gain access to the producer's position `kp`, which is implicitly stored in the closure representing

Specification of iteration methods.

$$\begin{array}{rcl} \mathit{isIter}\ \mathit{iter}\ \mathit{xs} & \triangleq & \square \forall\, I\ f. \\ & & \big(\forall\, \mathit{ys}\ z\ \mathit{zs}.\ \{I\ \mathit{ys}\ (z :: \mathit{zs})\}\ f\ z\ \{I\ (\mathit{ys}\ \texttt{@}\ [z])\ \mathit{zs}\}\big) \mathrel{-\!\!*} \{I\ []\ \mathit{xs}\}\ \mathit{iter}\ f\ \{I\ \mathit{xs}\ []\} \end{array}$$

Specification of lazy sequences.

$$\begin{array}{rcl} \mathit{isStream}\ S\ s\ \mathit{zs} & \triangleq & \{S\}\ s\ ()\ \{n.\ S * \mathit{isNode}\ S\ n\ \mathit{zs}\} \\ \mathit{isNode}\ S\ n\ \mathit{zs} & \triangleq & \texttt{case}\ n\ \texttt{of} \\ & & \quad \texttt{Nil => } \mathit{zs} = [] \\ & & \quad \texttt{| Cons (}z\texttt{, }s'\texttt{) => } \exists\, \mathit{zs}'.\ \mathit{zs} = z :: \mathit{zs}' * \mathit{isStream}\ S\ s'\ \mathit{zs}' \end{array}$$

Specification of `invert`.

$$\forall\, \mathit{iter}\ \mathit{xs}.\ \{\mathit{isIter}\ \mathit{iter}\ \mathit{xs}\}\ \texttt{invert}\ \mathit{iter}\ \{s.\ \exists\, S.\ S * \mathit{isStream}\ S\ s\ \mathit{xs}\}$$

Fig. 7. Specification of iteration methods, lazy sequences, and `invert`.

the `'a stream` for the next elements. When the consumer forces the thunk corresponding to this closure, control flows back to the producer at `kp` thanks to this recording mechanism. After (and if) `iter` terminates, the implementation jumps back with the value `Nil` to the consumer (whose position is stored in `r`) to signal the exhaustion of the collection.

Specification Figure 7 shows the specification of `invert`. The specification conforms to the informal reading that `invert` transforms an iteration method into a lazy sequence.

An iteration method is characterised in the logic via the assertion *isIter iter xs*, stating that *iter* iterates an input function over a collection whose elements are the same as those of the list *xs*.[6] The assertion *isIter iter xs* is defined in terms of a *loop invariant* I, a predicate over two lists of elements, a prefix and the remaining suffix of *xs*. It states that, for any input function f, if f is able to *process* any element z of the list, updating I from $I\ \mathit{ys}\ (z :: \mathit{zs})$ to $I\ (\mathit{ys}\ \texttt{@}\ [z])\ \mathit{zs}$, then *iter* f can process the entire list *xs*, updating I from $I\ []\ \mathit{xs}$ to $I\ \mathit{xs}\ []$.

A lazy sequence is characterised in the logic via the assertion *isStream S s zs*. The definition states that, when forced, the sequence s returns a node n such that *isNode S n zs* holds. The precondition S works as a permission to force s. Because it is recovered in the postcondition, it allows s to be called multiple times as long as the calls are sequential. Moreover, because it is existentially quantified in the postcondition of `invert`, it is abstract to a user of `invert`'s

[6] The use of a list in the characterisation of an iteration method *iter* restricts the order in which *iter* iterates its input function and rules out diverging iteration methods (in other words, the collection of elements must be finite). These simplifying assumptions can be lifted: Filliâtre and Pereira [9] show how possibly diverging iteration methods with non-specified iteration order can be characterised in a program logic. ↩

specification. The assertion $\mathit{isNode}\, S\, n\, zs$ states that n is either `Nil`, if zs is empty, or `Cons (`z`, `s'`)`, in which case zs is of the form z `::` zs' and s' is a sequence for zs', that is, the assertion $\mathit{isStream}\, S\, s'\, zs'$ holds.[7]

Verification There are two key steps in the verification of `invert`: (1) to find the permission S and (2) to find the loop invariant I maintained by the iteration *iter yield*, where *yield* denotes the value to which `yield` is bound (and r is the reference to which `r` is bound):

```
yield ≜ fn x => callcc (fn kp =>
          throw (valOf (!r)) (Cons (x, fn _ =>
            callcc (fn kc => (r := SOME kc; throw kp ()))
        )))
```

Both are, in fact, surprisingly simple:

$$\begin{aligned} S &\triangleq \exists\, y.\ r \mapsto y \\ I\ \_\ zs &\triangleq \exists\, kc.\ r \mapsto \texttt{SOME}\ kc\ *\ \mathit{isCont}\, kc\, (\lambda\, n.\ S * \mathit{isNode}\, S\, n\, zs) \end{aligned}$$

The permission S grants sequences ownership over the reference r.

The invariant I states that, whenever the elements in zs remain to be iterated, the reference r stores a continuation kc that corresponds to a consumer expecting a node n such that $\mathit{isNode}\, S\, n\, zs$ holds.

It is easy to see I `[]` xs holds at the beginning of *iter yield*, because r is initialised with a continuation kc supplied to the body of `invert` by a `callcc` expression whose postcondition Φ is precisely the predicate $\lambda\, n.\ S * \mathit{isNode}\, S\, n\, zs$. So, by Rule (Callcc), the assertion $\mathit{isCont}\, kc\, \Phi$ holds.

It is also easy to see that, if $I\ xs$ `[]` holds at the end of *iter yield*, then the final statement `throw (valOf (!`r`)) Nil` is correct, because the continuation kc stored in r would then expect a node n such that $\mathit{isNode}\, S\, n$ `[]` holds, thus requiring n to be `Nil`.

The crux then becomes to show that *yield* can process any element z of xs, updating the invariant accordingly:

$$\forall\, ys\, z\, zs.\ \{I\ ys\ (z\ \texttt{::}\ zs)\}\ \mathit{yield}\ z\ \{I\ (ys\ \texttt{@}\ [z])\ zs\}$$

This step follows mechanically from successive applications of our Rules (Callcc) and (Throw).

3.2 Dynamic Goto

Figure 8 shows the specification of `Dyngoto`'s components, namely the functions `new_lbl`, `checkpoint`, and `goto`. Their specifications are written in terms of *isLabel* assertions, which formalise the idea that a label records a program point. Indeed, the assertion $\mathit{isLabel}\, l\, \Phi$ states l stores a value `SOME` k where k

[7] By abuse of notation, we use `case` for a pattern matching at the level of the logic. ↩

Specification of `Dyngoto`*'s components.*

$$\{\mathit{True}\}\,\texttt{new\_lbl ()}\,\{l.\,\mathit{isLabel}\,l\,\_\}$$

$$\forall\, l\, e\, P\, \Phi.\ \begin{pmatrix}\{P * \mathit{isLabel}\,l\,\Phi\}\,e\,\{\Phi\} \mathrel{-\!\!*} \\ \{P * \mathit{isLabel}\,l\,\_\}\,\texttt{checkpoint}\,l\,(\texttt{fn \_ => }e)\,\{\Phi\}\end{pmatrix}$$

$$\forall\, l\, x\, \Phi.\ \{\mathit{isLabel}\,l\,\Phi * \Phi\, x\}\,\texttt{goto}\,l\,x\,\{\_.\,\mathit{False}\}$$

Definition of isLabel.

$$\begin{aligned}\mathit{isLabel}\,l\,\Phi &\triangleq \exists\, k.\ l \mapsto \texttt{SOME}\ k * \mathit{isCont}\,k\,\Phi \\ \mathit{isLabel}\,l\,\_ &\triangleq l \mapsto \texttt{NONE} \vee \exists\, \Phi.\ \mathit{isLabel}\,l\,\Phi\end{aligned}$$

Fig. 8. Specification of `Dyngoto`'s components and definition of *isLabel*.

is a continuation such that $\mathit{isCont}\,k\,\Phi$ holds. The assertion $\mathit{isLabel}\,l\,\_$, on the other hand, simply claims ownership of l; it is useful for stating the postcondition of `new_lbl` and the precondition of `checkpoint`.

The specification of `new_lbl` allows the introduction of new labels. The specification of `checkpoint` asserts that, to verify $\texttt{checkpoint}\,l\,(\texttt{fn \_ => }e)$ with postcondition Φ, it suffices to verify e with the same postcondition and under the assumption that $\mathit{isLabel}\,l\,\Phi$ holds. Under the interpretation of postconditions as program points, supplying the assumption $\mathit{isLabel}\,l\,\Phi$ corresponds with l being updated to store the program point in which the `checkpoint` appears. Finally, the specification of `goto` asserts that, if the label l stores a program point abstracted by Φ, and if x is a value such that $\Phi\, x$ holds, then it is safe to jump to l with x. The postcondition *False* is justified by the fact that `goto` never returns.

The verification of these specifications is straightforward given the definition of *isLabel*, the reasoning rules from Figure 2, and the following reasoning rule for a `case` statement when the scrutinee is a `SOME` value:

$$(\text{CaseSome})\ \frac{\{P\}\,(e\{x/\texttt{x}\})\,\{\Phi\}}{\{P\}\,\texttt{case SOME}\ x\ \texttt{of NONE => \_ | SOME x =>}\ e\,\{\Phi\}}$$

As an exercise, we invite the reader to verify the correctness of `fact` (§ 1) using the specifications from Figure 8 as principles to reason about `Dyngoto`.

4 Foundation

We use the logic *Maze* [28, Chapter 6] as the foundation for the reasoning principles from Figure 2. Maze is an Iris-based separation logic for effect handlers and multi-shot continuations. In Chapter 6.3.2 of de Vilhena's PhD thesis [28], he shows (1) how effect handlers and multi-shot continuations can be used to implement `callcc` and `throw` and (2) how Maze can then be used to derive reasoning principles for `callcc` and `throw`.

The Hoare triples from Figure 2 are encoded in Maze as follows:

$$\begin{aligned} \{P\}\, e\, \{\Phi\} &\triangleq \Box(P \mathrel{-\!\!*} \mathit{ewp}\, e\, \langle CT\rangle\, \{\Phi\}) \\ [P]\, e\, [\Phi] &\triangleq \Box(P \mathrel{-\!\!*} \mathit{ewp}\, e\, \langle \bot\rangle\, \{\Phi\}) \end{aligned}$$

The persistently modality is necessary to make the triples persistent as required by Rule (Persistent).

The predicate *ewp* is Maze's notion of the *weakest precondition*. The assertion $\mathit{ewp}\, e\, \langle \bot\rangle\, \{\Phi\}$ states that e either diverges or terminates with a value x such that $\Phi\, x$ holds. Additionally, the assertion $\mathit{ewp}\, e\, \langle CT\rangle\, \{\Phi\}$ allows e to execute `callcc` and `throw` statements according to a logical interface provided by the *protocol* CT. In a nutshell, a protocol allows the user of the logic to postulate the specification of an effect and to delay the soundness proof of this specification to the moment where this effect is given an implementation by an effect handler. In the case of CT, this protocol provides specifications to `callcc` and `throw` that respectively match Rules (Callcc) and (Throw). During the soundness proof of Maze, a toplevel effect handler providing operational meaning to `callcc` and `throw` becomes apparent and, by verifying this handler correct, we show that Rules (Callcc) and (Throw) hold with respect to `callcc` and `throw`'s semantics. For a deeper explanation of Maze, we refer the reader to Chapter 6 of de Vilhena's PhD thesis [28].

With this encoding, soundness of our Hoare triples follows immediately from soundness of Maze [28, Theorem 6.1]:

Theorem 1 (Soundness). *If the triple* $\{\mathit{True}\}\, e\, \{\_.\ \mathit{True}\}$ *is derivable using the principles from Figure 2, then* e *is* safe*; that is,* e *either diverges or terminates with a value.*

5 Related Work

Effect of `callcc` *to program reasoning.* As discussed in §1, Timany and Birkedal [27] show, via a counterexample, that the set of Hoare-style rules from Figure 1, which includes the bind rule and rules for `callcc` and `throw` that paraphrase their operational behaviour, is unsound.

Dreyer et al. [8] consider the effect of `callcc` to reasoning about program equivalence. Specifically, they study the effect of programming features to the validity of program equivalences according to two axes: (1) with versus without `callcc` and (2) *first-order state*, where only values of base types such as integers and Booleans can be stored, versus *higher-order state*, where any value can be stored including functions or continuations. They show that some program equivalences, namely the *awkward* example [22] and the *callback with lock* example [1], are lost when moving from a language without `callcc` to one that supports it. Interestingly, they also show that some of these equivalences, in particular, the callback with lock example, can be recovered when restricting the language to first-order state.

Program logics for `callcc`. Berger [3] introduces a program logic for an extension of PCF [23] with `callcc`, but without state. The logic possesses a rich assertion language in which program points (or *return ports* according to the terminology of the paper) can be manipulated via binders called *names* to specify the jumps in control flow. Berger shows that the logic enjoys *descriptive completeness* [14].

Crolard and Polonowski [4] introduce a program logic for reasoning about terminating programs with support for `callcc`, and for mutable stack variables, but no support for dynamically allocated references.

Separation logics for `callcc`. Delbiando and Nanevski [6] introduce a separation logic for `callcc` built on top of *Hoare Type Theory* [20]. As noted by Timany and Birkedal [27, § 9], their system does not support heap-stored continuations, a common idiom when programming with `callcc` that is used, for example, in the implementation of `invert` and `Dyngoto`.

Timany and Birkedal [27] develop a separation logic for `callcc` built on top of Iris [15]. Their logic admits the rules from Figure 1 except for the bind rule. Restricted support for the bind rule is enabled via a novel notion of weakest precondition called *context-local weakest precondition*, which admits the bind rule for the price of losing support for the rules for `callcc` and `throw` from Figure 1.

Data Availability

A Rocq formalisation of our results is available at `gitlab.inria.fr/cambium/hazel/blob/master/theories/case_studies/principles_of_callcc.v`.

Acknowledgements

I thank Xavier Leroy for his encouragement and for discussions that helped me refine the ideas in this paper. Moreover, I thank Amin Timany, Lars Birkedal, and the anonymous reviewers for their constructive feedback.

References

1. Ahmed, A., Dreyer, D., Rossberg, A.: State-Dependent Representation Independence. In: Principles of Programming Languages (POPL). pp. 340–353 (Jan 2009), `https://doi.org/10.1145/1480881.1480925`
2. Appel, A.W., MacQueen, D.B.: Standard ML of New Jersey. In: Programming Language Implementation and Logic Programming (PLILP). pp. 1–13 (Aug 1991). `https://doi.org/10.1007/3-540-54444-5_83`
3. Berger, M.: Program Logics for Sequential Higher-Order Control. In: Fundamentals of Software Engineering. Lecture Notes in Computer Science, vol. 5961, pp. 194–211 (Apr 2009), `https://doi.org/10.1007/978-3-642-11623-0_11`
4. Crolard, T., Polonowski, E.: Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. Journal of Logical and Algebraic Methods in Programming **81**(3), 181–208 (2012), `https://doi.org/10.1016/j.jlap.2012.01.004`

5. Cruanes, S., OCaml Contributors: OCaml Standard Library: Seq. `https://ocaml.org/api/Seq.html` (2025)
6. Delbianco, G.A., Nanevski, A.: Hoare-Style Reasoning with (Algebraic) Continuations. In: International Conference on Functional Programming (ICFP). pp. 363–376 (Sep 2013), `https://doi.org/10.1145/2544174.2500593`
7. Dijkstra, E.W.: Letters to the Editor: Go To Statement Considered Harmful. Communications of the ACM pp. 147–148 (Mar 1968), `https://doi.org/10.1145/362929.362947`
8. Dreyer, D., Neis, G., Birkedal, L.: The Impact of Higher-Order State and Control Effects on Local Relational Reasoning. In: International Conference on Functional Programming (ICFP). pp. 143–156 (Sep 2010), `https://doi.org/10.1145/1863543.1863566`
9. Filliâtre, J., Pereira, M.: A Modular Way to Reason About Iteration. In: NASA Formal Methods (NFM). Lecture Notes in Computer Science, vol. 9690, pp. 322–336 (Jun 2016), `https://doi.org/10.1007/978-3-319-40648-0_24`
10. Gabriel, R.: The Design of Parallel Programming Languages. `https://www.dreamsongs.com/10ideas.html` (1991)
11. Gansner, E.R., Reppy, J.H.: Standard ML Basis Library: The `List` structure. `https://smlfamily.github.io/Basis/list.html` (2004)
12. GHC Contributors: Haskell `base` Library: `Data.List`. `https://hackage.haskell.org/package/base/docs/Data-List.html` (2025)
13. Hewitt, C.: Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot (Apr 1972), AI Memo No. 251, MIT Project MAC
14. Honda, K., Berger, M., Yoshida, N.: Descriptive and Relative Completeness of Logics for Higher-Order Functions. In: Automata, Languages and Programming. pp. 360–371 (2006), `https://doi.org/10.1007/11787006_31`
15. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming (2018), `https://doi.org/10.1017/S0956796818000151`
16. Kiselyov, O.: An argument against `call/cc` (Dec 2012), `https://okmij.org/ftp/continuations/against-callcc.html`
17. Leroy, X.: You've got to decide either way or the other! – Classical logic, continuations, and control operators. `https://xavierleroy.org/CdF/2018-2019/4.pdf` (2018)
18. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: OCaml Standard Library: `List.iter`. `https://ocaml.org/api/List.html` (2025)
19. Madore, D.: A page about `call/cc`. `http://www.madore.org/~david/computers/callcc.html` (2002)
20. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare Type Theory, Polymorphism and Separation. Journal of Functional Programming **18**(5–6), 865–911 (2008), `https://doi.org/10.1017/S0956796808006953`
21. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Computer Science Logic. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (Sep 2001), `https://doi.org/10.1007/3-540-44802-0_1`
22. Pitts, A.M., Stark, I.D.B.: Operational Reasoning for Functions with Local State. In: Higher Order Operational Techniques in Semantics, p. 227–274 (1999), `https://homepages.inf.ed.ac.uk/stark/operfl.pdf`

23. Plotkin, G.D.: LCF Considered as a Programming Language. Theoretical Computer Science **5**(3), 225–255 (1977), https://doi.org/10.1016/0304-3975(77)90044-5
24. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002), https://www.cs.cmu.edu/~jcr/seplogic.pdf
25. Standard ML of New Jersey: SMLofNJ.Cont. https://www.smlnj.org/doc/SMLofNJ/pages/cont.html
26. Stepanenko, S., Nardino, E., Frumin, D., Timany, A., Birkedal, L.: Context-Dependent Effects in Guarded Interaction Trees. In: Programming Languages and Systems. pp. 286–313 (2025). https://doi.org/10.1007/978-3-031-91121-7_12
27. Timany, A., Birkedal, L.: Mechanized Relational Verification of Concurrent Programs with Continuations. Proceedings of the ACM on Programming Languages **3**(ICFP), 105:1–105:28 (Jul 2019), https://doi.acm.org/10.1145/3341709
28. de Vilhena, P.E.: Proof of Programs with Effect Handlers. Ph.D. thesis, Université Paris Cité (Dec 2022), https://inria.hal.science/tel-03891381