

## Getting There and Back Again

**Olivier Danvy\***

*Yale-NUS College & School of Computing*  
*National University of Singapore*  
 danvy@acm.org

**Abstract.** “There and Back Again” (TABA) is a programming pattern where the recursive calls traverse one data structure and the subsequent returns traverse another. This article presents new TABA examples, refines existing ones, and formalizes both their control flow and their data flow using the Coq Proof Assistant. Each formalization mechanizes a pen-and-paper proof, thus making it easier to “get” TABA. In addition, this article identifies and illustrates a tail-recursive variant of TABA, There and Forth Again (TAFE) that does not come back but goes forth instead with more tail calls.

**Keywords:** TABA, recursion pattern, list processing, symbolic convolutions, eta redexes, Coq Proof Assistant, There and Forth Again (TAFE), defunctionalization and refunctionalization, lambda-lifting and lambda-dropping, lightweight fusion and lightweight fission, continuations.

Dear Reader:

Unless you are already acquainted with “There and Back Again,” could you first spend a few minutes thinking about the following programming exercises before proceeding any further? Thank you.

**Convolving a list with itself:**

Given a list  $[x_1, x_2, \dots, x_{n-1}, x_n]$ , where  $n$  is unknown, construct  $[(x_1, x_n), (x_2, x_{n-1}), \dots, (x_{n-1}, x_2), (x_n, x_1)]$  in  $n$  recursive calls. The implementation should be expressible using a fold function for lists.

\*Address for correspondence: Yale-NUS College & School of Computing, National University of Singapore.

**Convolving a list with itself (continued):**

Here is a non-solution in OCaml:

```
let self_convolve xs = (* 'a list -> ('a * 'a) list *)
  List.map2 (fun x x_op -> (x, x_op)) xs (List.rev xs);;
```

(This implementation is not a solution because it incurs two independent traversals: one for reversing the second list (with `List.rev`), and one for zipping the given list and its reverse (with `List.map2`.)

**Deciding whether two lists are reverses of each other:**

Given two lists of unknown lengths, test whether one list is the reverse of the other, if they have the same length  $n$ . Each list should be traversed only once, no intermediate list should be created, and the implementation should proceed in  $n$  recursive calls, i.e., be expressible using a fold function. Here is a non-solution in OCaml:

```
let rev2 vs ws =
  vs = List.rev ws;;
```

(This implementation is not a solution because it incurs two independent traversals: one for reversing the second list (with `List.rev`), and one for comparing the first list and the reversed second list (with `=`.)

**Deciding whether a lambda term has the shape of an eta redex:**

Given the abstract-syntax tree of a  $\lambda$  term, test whether it is shaped like the  $\eta$  redex  $\lambda x_1.\lambda x_2.\dots\lambda x_n.e x_1 x_2 \dots x_n$ , where  $e$  and  $n$  are unknown, in  $n$  recursive calls.

**Indexing a list from right to left:**

Given a list and a non-negative integer, index this list from the right:

```
let test_list_index_rtl candidate =
  (candidate [3; 2; 1; 0] 0 = Some 0) &&
  (candidate [3; 2; 1; 0] 3 = Some 3) &&
  (candidate [3; 2; 1; 0] 4 = None);;
```

The list should be traversed only once and no intermediate list should be created. Here are two non-solutions in OCaml:

```
let list_index_rtl_rev_list vs n = (* 'a list -> int -> 'a option *)
  if 0 <= n && n < List.length vs
  then Some (List.nth (List.rev vs) n)
  else None;;

let list_index_rtl_rev_index vs n = (* 'a list -> int -> 'a option *)
  if 0 <= n
  then let length_vs = List.length vs
       in if n < length_vs
          then let last_index = pred length_vs
               in Some (List.nth vs (last_index - n))
          else None
  else None;;
```

**Indexing a list from right to left (continued):**

(The first implementation passes the unit test but is not a solution because it independently traverses the given list three times: once for computing its length (with `List.length`), once for reversing it (with `List.rev`), and once for indexing it from left to right (with `List.nth`). Also, it constructs an intermediate list (with `List.rev`). The second implementation also passes the unit test but is not a solution either because it independently traverses the given list twice: once for computing its length and once for indexing it from left to right.)

## 1. Background and introduction

“There and Back Again” (TABA for short [1]) was a new programming pattern of structural recursion where one data structure is traversed at call time (as usual) and another at return time (which was new). This pattern makes it possible, for example, to symbolically convolve the lists  $[v_1; v_2; \dots; v_{n-1}; v_n]$  and  $[w_1; w_2; \dots; w_{n-1}; w_n]$  into  $[(v_1, w_n); (v_2, w_{n-1}); \dots; (v_{n-1}, w_2); (v_n, w_1)]$  in one recursive traversal of either list at call time and one iterative traversal of the other list at return time, without constructing any intermediate list in reverse order. Other examples include, e.g., multiplying polynomials, computing Catalan numbers, testing whether a given list is a palindrome, and testing whether a given abstract-syntax tree is a generalized  $\beta$  redex without constructing any intermediate data structure. All of these examples are therefore expressible using primitive iteration, i.e., a fold function.

The essence of TABA is that traversing a data structure recursively at call time accumulates enough computing power to traverse another data structure iteratively at return time. This second traversal can be either implicit in direct style—which at first sight may feel as unusual as seeing Charlie Chaplin or Marcel Marceau walking against the wind or someone performing the moonwalk dance—or explicit in continuation-passing style, where the continuation is delimited and takes the form of an iterator. Defunctionalizing this continuation gives rise to the intermediate data structure that is explicitly iterated upon and that one would naturally use in a tail-recursive solution, paving the way to constructing TABA examples by refunctionalizing non-solutions [2].

By now, TABA is often cited as a programming pattern that illustrates recursion and continuations [3, 4, 5, 6]. It has inspired a new calculation rule, IO swapping [7], and it has proved useful to express filters in XML [8], to illustrate advances in type inference [9], to process tail-aligned lists [10], to illustrate introspection [11], and to carry out backpropagation compositionally [12, 13, 14]. TABA programs have been implemented in many languages, from Prolog and C to Agda. At least two have been formalized in Why3 [15].

In this article, we present new TABA examples and we suggest a variation that simplifies the original treatment of convolutions to handle the pathological case where the two lists to convolve do not have the same length. This new convolving function traverses both given lists at call time instead of only one of them and its continuation is only applied when these two lists have the same length. The codomain of this continuation then no longer needs to be an option type to accommodate the

pathological case. The new convolving function is still primitive iterative and thus it can be expressed using a fold function, whether in direct style or in continuation-passing style. We also identify a tail-recursive variant of TABA, “There and Forth Again” (TAFE), where there is no going back.

All of the above was formalized with the Coq Proof Assistant [16]. For clarity, the proofs use no automation and correspond to what one would traditionally write by hand [17, 18, 19]. Also, they are all equational, even for implementations that use a continuation.

The rest of this article is structured as follows. We start by solving the first exercise in the opening “Dear Reader” box, namely we implement a function that convolves any given list of (unknown) length  $n$  with itself in  $n$  recursive calls (Section 2). We then treat another simple example of TABA that is also new: given two lists of unknown length, we determine whether one is the reverse of the other, without reversing either list (Section 3). As for the pathological case where the two lists do not have the same length, we propose to handle it a priori (i.e., at call time) instead of a posteriori (i.e., at return time), which simplifies both the program and its proof (Section 3.6 and beyond). We then revisit the canonical example of TABA, convolving two lists (Section 4). Turning to the third exercise in the opening “Dear Reader” box, we show how to detect whether a lambda term has the shape of an eta redex in one recursive descent of its abstract-syntax tree and without constructing an intermediate data structure (Section 5). Solving the last exercise in the opening “Dear Reader” box suggests a “There and Forth Again” programming pattern that is a tail-recursive variant of TABA in the case where the order in which to traverse the data structure at return time does not matter (Section 6). We then return to convolving two lists that may not have the same length (Section 7) before concluding (Section 8) and closing with another “Dear Reader” box containing more programming exercises.

**Prerequisites and notations:** This article assumes an elementary knowledge of OCaml and of the Coq Proof Assistant (and its pure and total functional programming language Gallina). It also hinges on an awareness of continuations and defunctionalization, which are reviewed in the appendix. Notationally, the Haskell plural convention is used throughout: if  $v$  denotes a value,  $vs$  denotes a list of values and if  $p$  denotes a pair,  $ps$  denotes a list of pairs.

## 2. Convolving a list with itself

The goal of this section is to implement a function that, given a list, convolves this list with itself. For example, in OCaml, this function maps `[1; 2; 3; 4]` to `[(1, 4); (2, 3); (3, 2); (4, 1)]`. The result is a symbolic self-convolution of the given list because enumerating the first projections of its successive pairs yields the given list, and enumerating their second projections yields the reverse of the given list. (Both enumerations are achieved using the `unzip` function. By that book, the `zip` function achieves a symbolic dot product.)

We first program this function in OCaml recursively using the TABA programming pattern and illustrate this pattern with a trace (Section 2.1) before formalizing this implementation using the Coq Proof Assistant (Section 2.2). We then program this function in OCaml tail recursively with continuations using the TABA programming pattern and illustrate this pattern with a trace (Section 2.3) before formalizing this implementation using the Coq Proof Assistant (Section 2.4).

## 2.1. First-order programming

The following implementation uses the TABA programming pattern in that its auxiliary function traverses the given list (denoted by *vs*) at call time and then traverses the given list (denoted by *ws*) again at return time, constructing the resulting list of pairs (denoted by *ps*):

```
let self_cnv vs = (* 'a list -> ('a * 'a) list *)
  let rec visit vs_sfx = (* 'a list -> 'a list * ('a * 'a) list *)
    match vs_sfx with
    | [] -> (vs, [])
    | v :: vs_sfx' -> let (ws, ps) = visit vs_sfx'
                      in (List.tl ws, (v, List.hd ws) :: ps)
  in let (_, ps) = visit vs
     in ps;;
```

where “cnv” stands for “convolution” in the name of this function.

Let us visualize the induced computation with a trace:

- when a function is called, its name and its argument(s) are printed and followed by a right arrow;
- when a function returns, its name and its argument(s) are printed and followed by a left arrow, itself followed by the result that is being returned; and
- the printed names are indented to reflect the nested calls, in the tradition of tracing in Lisp.

To wit:

```
# traced_self_cnv show_int [1; 2; 3];;
self_cnv [1; 2; 3] ->
  visit [1; 2; 3] ->
    visit [2; 3] ->
      visit [3] ->
        visit [] ->
          visit [] <- ([1; 2; 3], [])
          visit [3] <- ([2; 3], [(3, 1)])
          visit [2; 3] <- ([3], [(2, 2); (3, 1)])
          visit [1; 2; 3] <- ([], [(1, 3); (2, 2); (3, 1)])
self_cnv [1; 2; 3] <- [(1, 3); (2, 2); (3, 1)]
- : (int * int) list = [(1, 3); (2, 2); (3, 1)]
#
```

In this example, the self-convolution function is called with the list `[1; 2; 3]`, and then calls the `visit` function recursively as it traverses this list. That the given list is traversed is visualized by the successive calls to the `visit` function, and that the calls are nested is rendered by the indentations. When the end of the list is reached, a pair is returned that contains the given list and an empty list of pairs, which initiates a series of returns that matches the series of calls. That the given list is traversed is visualized by the successive returns from the `visit` function, and that the returns match the calls is rendered by the vertically aligned indentations. The list of pairs is constructed (1) using the head of the list that was accessed at call time and that is still available in the lexical environment and (2) using the head of the returned list. After the last return, the returned list is empty and the list of pairs is the symbolic convolution.

## 2.2. Formalizing and proving the first-order program

The implementation above cannot directly be formalized in the Coq Proof Assistant because of the accessors `List.hd` and `List.tl` that optimistically assume that they are applied to a non-empty list (and otherwise raise an error). This optimistic assumption actually holds, but the type system is not strong enough to account for it, and so we have to use an option type, which incurs a proof obligation that the implementation is total, i.e., always returns a result constructed with `Some`. We also lambda-lift the implementation (see Appendix C), declaring the auxiliary function `visit` globally too instead of locally, to reason about it using its global name:

```
Fixpoint self_cnv' (V : Type) (vs_sfx vs : list V) : option (list V * list (V * V)) :=
  match vs_sfx with
  | nil          => Some (vs, nil)
  | v :: vs_sfx' => match self_cnv' V vs_sfx' vs with
    | Some (ws, ps) => match ws with
      | nil          => None (* impossible case *)
      | w :: ws'    => Some (ws', (v, w) :: ps)
    | end
    | None          => None
  | end
end.
```

The codomain uses an option type to account for the impossible case where the returned list does not have the same length as the given list (too short above, too long below):

```
Definition self_cnv (V : Type) (vs : list V) : option (list (V * V)) :=
  match self_cnv' V vs vs with
  | Some (ws, ps) => match ws with
    | nil          => Some ps
    | w :: ws'    => None (* impossible case *)
  | end
  | None          => None
end.
```

The list returned by `self_cnv'`, `ws`, is supposed to be empty. The result is the returned list of pairs, `ps`.

The following theorem captures that the implementation is sound and complete as well as total:

```
Theorem soundness_and_completeness_of_self_cnv :
  forall (V : Type) (vs : list V) (ops : option (list (V * V))),
    self_cnv V vs = ops <-> exists ps : list (V * V),
      ops = Some ps /\ map fst ps = vs /\ map snd ps = rev vs.
```

where `fst` and `snd` respectively denote the first and the second pair projections. In words – `self_cnv` is total in that it always maps a list to an optional list of pairs that is constructed with `Some`; it is sound in that enumerating the first projections of the resulting list of pairs coincides with the given list and enumerating the second projections of this list of pairs coincides with the reverse of the given list, which is the definition of a symbolic self-convolution; and it is complete in that given the symbolic self-convolution `ps` of a list `vs`, applying `self_cnv` to `vs` totally yields `ps`.

This theorem is a corollary of the following lemma about `self_cnv'` which captures the essence of TABA's control flow and of TABA's data flow:

```

Lemma about_self_cnv' :
  forall (V : Type) (vs_sfx ws_pfx : list V),
    length vs_sfx = length ws_pfx ->
    forall ws_sfx : list V,
      exists ps : list (V * V),
        self_cnv' V vs_sfx (ws_pfx ++ ws_sfx) = Some (ws_sfx, ps) /\
        map fst ps = vs_sfx /\ map snd ps = rev ws_pfx.

```

The second argument of `self_cnv'`, `vs_sfx`, is the list traversed by the calls so far and its third argument, `vs`, is the given list. So `vs_sfx` is a suffix of `vs`. In this lemma, this given list is also expressed as the concatenation of a prefix, `ws_pfx`, and a suffix, `ws_sfx`, which are such that the length of this prefix is the same as the length of the list traversed so far, `vs_sfx`.

**Control flow:** The lemma expresses how the given list is traversed at return time: the lengths of `vs_sfx` and of `ws_pfx` are the number of remaining calls to traverse `vs_sfx`. By the very nature of structural recursion, this number is also the number of returns that yield `Some (ws_sfx, ps)`. Therefore the returns have traversed the current prefix of the given list, `ws_pfx`, and the returned list is its current suffix, `ws_sfx`.

**Data flow:** The lemma also captures that the returned list of pairs is a symbolic convolution of the list that remains to be traversed at call time, namely `vs_sfx`, and of the list that has been traversed at return time, namely `ws_pfx`.

The control-flow aspect of the lemma expresses that `ws_pfx` has been traversed by the returns and that `vs_sfx` remains to be traversed by the calls. The data-flow aspect of the lemma expresses that the returned list of pairs is a symbolic convolution of these two lists.

The lemma is proved by induction on `vs_sfx`. In the course of this proof, the following property came handy to connect the first element of a non-empty list and its following suffix and the last element of this non-empty list and its preceding prefix:

```

Property from_first_and_suffix_to_prefix_and_last :
  forall (V : Type) (v : V) (vs : list V),
    exists (ws : list V) (w : V),
      v :: vs = ws ++ w :: nil /\ length vs = length ws.

```

This property is proved by induction on `vs`.

## 2.3. Higher-order programming

The following implementation uses the TABA programming pattern in that its auxiliary function traverses the given list at tail-call time and accumulates a two-argument continuation to traverse the given list again and construct the resulting list of pairs:

```

let self_cnv_c vs = (* 'a list -> ('a * 'a) list *)
  let rec visit vs_sfx k = (* 'a list -> ('a list -> ('a * 'a) list) -> ('a * 'a) list *)
    match vs_sfx with
    | [] -> k vs []
    | v :: vs_sfx' -> visit vs_sfx' (fun ws ps -> k (List.tl ws) ((v, List.hd ws) :: ps))
  in visit vs (fun _ ps -> ps);;

```

Let us visualize the induced computation with a trace:

```
# traced_self_cnv_c show_int [1; 2; 3];;
self_cnv_c [1; 2; 3] ->
visit [1; 2; 3] continuation_0 ->
visit [2; 3] continuation_1 ->
visit [3] continuation_2 ->
visit [] continuation_3 ->
continuation_3 [1; 2; 3] [] ->
continuation_2 [2; 3] [(3, 1)] ->
continuation_1 [3] [(2, 2); (3, 1)] ->
continuation_0 [] [(1, 3); (2, 2); (3, 1)] ->
- : (int * int) list = [(1, 3); (2, 2); (3, 1)]
#
```

In this example, the self-convolution function is called with the list `[1; 2; 3]`, and then calls the `visit` function tail-recursively as it traverses this list and accumulates a continuation. That the given list is traversed is visualized by the successive calls to the `visit` function, and that the calls are tail calls is rendered by the lack of indentation. When the end of the list is reached, the current continuation is passed the given list and an empty list of pairs, and then tail-calls the accumulated continuations in a way that matches the series of tail-calls to `visit`. That the given list is traversed is visualized by the successive tail-calls to the continuations, and that these tail-calls match the corresponding calls is rendered with the name of these continuations. The list of pairs is constructed (1) using the head of the list that was accessed at tail-call time and that is still available in the lexical environment and (2) using the head of the list that is passed to the continuation. When the initial continuation is passed a list and a list of pairs, the former list is empty and the latter list is the symbolic convolution.

## 2.4. Formalizing and proving the higher-order program

As in Section 2.2, using the partial functions `List.hd` and `List.tl` is not an option in the Coq Proof Assistant, and so we need to use an option type in the codomain, which also incurs a proof obligation that the implementation is total. Likewise, the implementation is lambda-lifted and so the local function is defined globally too, as a recursive equation:

```
Fixpoint self_cnv_c' (V : Type)
  (vs_sfx vs : list V)
  (k : list V -> list (V * V) -> option (list (V * V)))
  : option (list (V * V)) :=
  match vs_sfx with
  | nil =>
  | k vs nil
  | v :: vs_sfx' =>
    self_cnv_c' V vs_sfx' vs (fun ws ps =>
      match ws with
      | nil => None (* impossible case *)
      | w :: ws' => k ws' ((v, w) :: ps)
      end)
  end.
```



```

Definition self_cnv_c (V : Type) (vs : list V) : option (list (V * V)) :=
  self_cnv_c' V vs vs (fun ws ps => match ws with
    nil      => Some ps
  | w :: ws' => None (* impossible case *)
  end).

```

The initial continuation expects a list and a list of pairs. This list is supposed to be empty, and the result is this list of pairs.

As in Section 2.2, the following theorem captures soundness, completeness, and totality:

```

Theorem soundness_and_completeness_of_self_cnv_c :
  forall (V : Type) (vs : list V) (ops : option (list (V * V))),
    self_cnv_c V vs = ops <-> exists ps : list (V * V),
      ops = Some ps /\ map fst ps = vs /\ map snd ps = rev vs.

```

Similarly to Section 2.2, this theorem is a corollary of the following lemma about `self_cnv_c'` which captures the essence of TABA's control flow and of TABA's data flow:

```

Lemma about_self_cnv_c' :
  forall (V : Type) (vs_sfx ws_pfx : list V),
    length vs_sfx = length ws_pfx ->
  forall (ws_sfx : list V)
    (k : list V -> list (V * V) -> option (list (V * V))),
    exists ps : list (V * V),
      self_cnv_c' V vs_sfx (ws_pfx ++ ws_sfx) k = k ws_sfx ps /\
      map fst ps = vs_sfx /\ map snd ps = rev ws_pfx.

```

In Section 2.2, `self_cnv'` returns `Some (ws_sfx, ps)` whereas `self_cnv_c'` continues with `ws_sfx` and `ps` here. This lemma is also proved by induction on `vs_sfx`.

## 2.5. Summary, synthesis, and significance

Through functional programming and proving, we have illustrated the TABA programming pattern with one simple example, the symbolic self-convolution of a list. The illustration was both visual, using a trace of the successive calls and returns, and logical, with a lemma that characterizes both the control flow and the data flow of the TABA programming pattern, be it recursive or tail-recursive with continuations.

## 3. Testing whether two lists are reverses of each other

The goal of this section is to implement a predicate that decides whether two given lists of unknown length are reverses of each other. We are going to inter-derive a spectrum of polymorphic functions `rev2 : forall V : Type, (V -> V -> bool) -> list V -> list V -> bool`, each of which satisfies the following theorem:

```

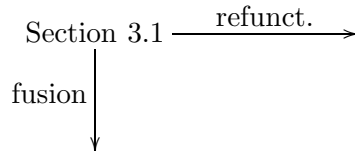
Theorem soundness_and_completeness_of_rev2 :
  forall (V : Type) (beq_V : V -> V -> bool),
    (forall v v' : V, beq_V v v' = true <-> v = v') ->
  forall vs_given ws_given : list V,
    rev2 V beq_V vs_given ws_given = true <-> rev vs_given = ws_given.

```

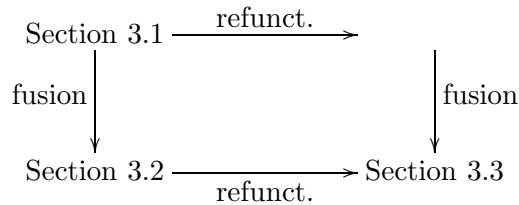
In words – given a type  $V$  of comparable values (and an associated equality predicate  $\text{beq}_V$  that is sound and complete),  $\text{rev2}$  is sound in that if applying it to two lists yields  $\text{true}$ , then the two lists are reverses of each other, and it is complete in that applying it to two lists that are reverses of each other yields  $\text{true}$ .

In each of the implementations,  $\text{rev2}$  is a main function that uses one or two auxiliary functions. Accordingly, the main theorem is a corollary of auxiliary lemmas that we will state. Some of these auxiliary lemmas are about the soundness and completeness of the auxiliary functions, and some others are about a useful property of these auxiliary functions.

We first start from the “non-solution” that reverses the first list and then traverses the reversed first list together with the second list (Section 3.1). As it happens, this non-solution is a candidate both for lightweight fusion (Appendix D) and for refunctionalization (Appendix A):

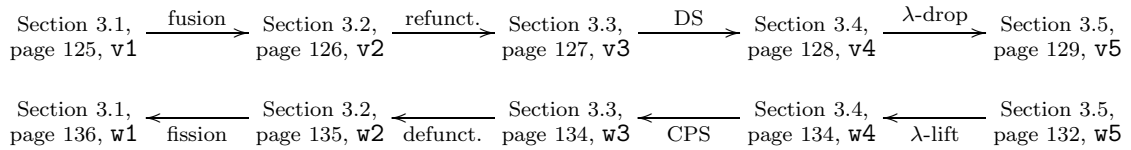


We successively lightweight-fuse it (Section 3.2) and then refunctionalize the result (Section 3.3), but the converse would do as well, as the end result is the same:



As it happens again, this end result is both structurally recursive and thus expressible using a fold function (Appendix B) and in continuation-passing style (Appendix E). We map it back to direct style (Section 3.4) and then lambda-drop the result (Appendix C) from two recursive equations to one block-structured, lexically scoped program (Section 3.5).

We then turn to the pathological case where the two given lists do not have the same length. We simplify this case by treating it at call time rather than at return time, i.e., by traversing both lists at call time instead of only one of them, and by only returning if the two given lists have the same length (Section 3.6). Henceforth, we adjust the block-structured, lexically scoped program, and then lambda-lift it into two recursive equations, CPS-transform them into a continuation-passing program, and then defunctionalize and defuse this program into a non-solution that reverses the first list and then traverses the reversed first list together with the second list if they have the same length. And at each step, we state the corresponding auxiliary lemmas. Diagrammatically, and with an unambiguous abuse of notation – the section numbers are duplicated [20] – this spectrum can be rendered as follows:



In both Sections 3.2, the program is tail recursive and implements a pushdown automaton where the intermediate list acts as the stack. In both Sections 3.3, the program is tail recursive and the stack is represented as an explicit continuation. And in both Sections 3.4, the program is not tail recursive and the explicit continuation is represented by the implicit control stack that underlies language processors since Dijkstra [21] to implement nested calls in general and recursive calls in particular, an implementation technique which is so salient that it is used nowadays to *explain* recursion. The continuation-passing programs in Sections 3.3 and the direct-style programs in Sections 3.4 and 3.5 also illustrate TABA in that one of the lists is traversed at call time and the other at return time. These programs are structurally recursive, or more precisely they are primitive iterative in that they can be expressed with a fold function: they can therefore be reasoned about using structural induction.

### 3.1. A first-order implementation in two passes

Let us start from the non-solution below:

1. The first list is reversed, tail recursively with an accumulator, using `rev2''_v1`.
2. The reversed first list and the second list are traversed together, tail recursively, using `rev2''_v1`. The result is a Boolean: either these two lists have the same length and the same elements, i.e., are structurally equal, or they do not.

In Gallina:

```
Fixpoint rev2''_v1 (V : Type) (beq_V : V -> V -> bool) (vs_op ws : list V) : bool :=
  match vs_op with
  | nil => match ws with
    | nil => true
    | w :: ws' => false
    end
  | v :: vs'_op => match ws with
    | nil => false
    | w :: ws' => if beq_V v w
      then rev2''_v1 V beq_V vs'_op ws'
      else false
    end
  end.
```

```
Fixpoint rev2'_v1 (V : Type) (vs vs_op : list V) : list V :=
  match vs with
  | nil => vs_op
  | v :: vs' => rev2'_v1 V vs' (v :: vs_op)
  end.
```

```
Definition rev2_v1 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  rev2''_v1 V beq_V (rev2'_v1 V vs_given nil) ws_given.
```

The two auxiliary definitions give rise to two auxiliary lemmas, a corollary of which is the soundness and completeness of `rev2_v1`. The first is proved by induction on `vs_op` and the second by induction on `vs`:

```
Lemma soundness_and_completeness_of_rev2''_v1 :
  forall (V : Type) (beq_V : V -> V -> bool),
    (forall v v' : V, beq_V v v' = true <-> v = v') ->
    forall vs_op ws : list V,
      rev2''_v1 V beq_V vs_op ws = true <-> vs_op = ws.
```

```
Lemma soundness_and_completeness_of_rev2'_v1 :
  forall (V : Type) (vs vs_op vs_given_op : list V),
    rev2'_v1 V vs vs_op = vs_given_op <-> rev vs ++ vs_op = vs_given_op.
```

In words – `rev2''_v1` is a sound and complete implementation of structural equality, and `rev2'_v1` is a sound and complete implementation of list reversal with an accumulator. (To say it again, soundness means that if the implementation yields a result, this result is correct, and completeness means that if a result is expected, then the implementation provides it.)

### 3.2. A first-order and tail-recursive implementation

The goal of lightweight fusion by fixed-point promotion [22] is to make an implementation tail recursive, and the implementation in Section 3.1 is a fitting case for it: in the definition of `rev2_v1`, the tail-recursive function `rev2'_v1` is called, and its result is used to call `rev2''_v1`. Lightweight fusion relocates the context of the call to `rev2'_v1` from the definition of `rev2_v1` to the definition of `rev2'_v1`, making it not return, but instead perform a tail call to `rev2''_v1`:

```
Fixpoint rev2'_v2 (V : Type) (beq_V : V -> V -> bool) (vs vs_op ws_given : list V) : bool :=
  match vs with
  | nil => rev2''_v1 V beq_V vs_op ws_given
  | v :: vs' => rev2'_v2 V beq_V vs' (v :: vs_op) ws_given
  end.
```

```
Definition rev2_v2 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  rev2'_v2 V beq_V vs_given nil ws_given.
```

In words – whereas `rev2'_v1` was called non-tail recursively in the definition of `rev2_v1`, which then tail-called `rev2''_v1`, `rev2'_v2` is called tail recursively in the definition of `rev2_v2`, which then tail-calls `rev2''_v1`.

The auxiliary definition above gives rise to an auxiliary lemma, a corollary of which is the soundness and completeness of `rev2_v2`:

```
Lemma about_rev2'_v2 :
  forall (V : Type) (beq_V : V -> V -> bool) (vs vs_op ws_given : list V),
    exists vs_given_op : list V,
      rev2'_v2 V beq_V vs vs_op ws_given = rev2''_v1 V beq_V vs_given_op ws_given /\
      rev vs ++ vs_op = vs_given_op.
```

In words – `rev2'_v2` ends up tail-calling `rev2''_v1` on the reverse of the first list. This lemma is proved by induction on `vs`.

### 3.3. A higher-order and tail-recursive implementation

The implementation in Section 3.2 is in defunctionalized form (see Appendix A) in that the intermediate list (i.e., `vs_op`), together with the second pass (i.e., `rev2''_v1`) that consumes it, can be represented as a function, `h_vs_op`:

```
Fixpoint rev2'_v3 (V : Type) (beq_V : V -> V -> bool)
  (vs : list V) (h_vs_op : list V -> bool) (ws_given : list V) : bool :=
  match vs with
  | nil => h_vs_op ws_given
  | v :: vs' => rev2'_v3 V beq_V vs' (fun ws => match ws with
    | nil => false
    | w :: ws' => if beq_V v w
      then h_vs_op ws'
      else false
    end) ws_given
  end.
```

```
Definition rev2_v3 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  rev2'_v3 V beq_V vs_given (fun ws => match ws with
  | nil => true
  | w :: ws' => false
  end) ws_given.
```

In words – in Sections 3.1 and 3.2, the elements of the first list were accumulated into a list in reverse order (`vs_op`) and then this list was traversed together with the second list to check for structural equality in `rev2''_v1`. Here, the elements of the first list are accumulated into a function (`h_vs_op`) to traverse the second list and then this function is applied to the second list to carry out this traversal.

The auxiliary definition above gives rise to two auxiliary lemmas, a corollary of which is the soundness and completeness of `rev2_v3`:

```
Lemma soundness_of_rev2'_v3 :
  forall (V : Type) (beq_V : V -> V -> bool),
    (forall v v' : V, beq_V v v' = true -> v = v') ->
    forall (vs : list V) (h_vs_op : list V -> bool) (ws_given : list V),
      (exists ws : list V,
        rev2'_v3 V beq_V vs h_vs_op ws_given = h_vs_op ws /\ rev vs ++ ws = ws_given) \/
        rev2'_v3 V beq_V vs h_vs_op ws_given = false.
```

In words – either `rev2'_v3` ends up tail-calling `h_vs_op` on a suffix of `ws_given` such that the corresponding prefix is the reverse of `vs` or it yields `false`. This soundness lemma is proved by induction on `vs`.

```
Lemma completeness_of_rev2'_v3 :
  forall (V : Type) (beq_V : V -> V -> bool),
    (forall v v' : V, v = v' -> beq_V v v' = true) ->
    forall vs ws_pfx ws_sfx ws_given : list V,
      rev vs = ws_pfx ->
      ws_pfx ++ ws_sfx = ws_given ->
      forall h_vs_op : list V -> bool,
        rev2'_v3 V beq_V vs h_vs_op ws_given = h_vs_op ws_sfx.
```

In words – if  $vs$  is the reverse of a prefix of  $ws\_given$ , then  $rev2'\_v3$  ends up tail-calling  $h\_vs\_op$  on the corresponding suffix. This completeness lemma is proved by induction on  $vs$ .

Lightweight-fusing Version 1 and then refunctionalizing the result (as done above) or refunctionalizing Version 1 and then lightweight-fusing the result yield the same result—and what a remarkable result it is: an implementation which is (1) structurally recursive and therefore can be expressed using fold-right (see Appendix B), and (2) in continuation-passing style (see Appendix E) where all calls are tail calls and  $h\_vs\_op$  acts as the continuation. This implementation is characteristic of TABA: the first list is traversed at call time (when the continuation is accumulated) and the second at return time (when the continuation is applied).

### 3.4. A first-order and recursive implementation, lambda-lifted

In the implementation of Section 3.3 the continuation is delimited: it is initialized in  $rev2\_v3$ , and it is only applied in  $rev2'\_v3$  if the second list is long enough and its current first element coincides with the corresponding current first element of the first list; otherwise, the current continuation is not applied and the computation is discontinued. In other words, this implementation can be expressed in direct style with an exception to account for the current continuation not being applied.

Gallina being a pure functional language, it does not feature exceptions. To express this implementation in direct style, we first need to encode the exceptional behavior using an option type, which makes the continuation linear (each continuation is defined once and used once, last in, first out). The corresponding direct-style implementation reads as follows:

```
Fixpoint rev2'_v4 (V : Type) (beq_V : V -> V -> bool)
  (vs : list V) (ws_given : list V) : option (list V) :=
  match vs with
  | nil      => Some ws_given
  | v :: vs' => match rev2'_v4 V beq_V vs' ws_given with
    | Some ws => match ws with
      | nil      => None
      | w :: ws' => if beq_V v w then Some ws' else None
    | None     => None
  end
end
end.
```

```
Definition rev2_v4 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  match rev2'_v4 V beq_V vs_given ws_given with
  | Some ws => match ws with
    | nil      => true
    | w :: ws' => false
  | None     => false
end.
```

This implementation is characteristic of TABA: the successive calls to  $rev2'\_v4$  traverse the first list and its successive returns traverse the second, without creating any intermediate list. In case of mismatch,  $None$  is incrementally returned until the initial call to  $rev2'\_v4$  in the definition of  $rev2\_v4$ .

The auxiliary definition gives rise to two auxiliary lemmas, a corollary of which is the soundness and completeness of `rev2_v4`. These lemmas are proved by induction on `vs`:

```

Lemma soundness_of_rev2'_v4 :
forall (V : Type) (beq_V : V -> V -> bool),
  (forall v v' : V, beq_V v v' = true -> v = v') ->
  forall vs ws_given : list V,
    (exists ws : list V,
      rev2'_v4 V beq_V vs ws_given = Some ws /\ rev vs ++ ws = ws_given) \/
      rev2'_v4 V beq_V vs ws_given = None.

Lemma completeness_of_rev2'_v4 :
forall (V : Type) (beq_V : V -> V -> bool),
  (forall v v' : V, v = v' -> beq_V v v' = true) ->
  forall vs ws ws_given : list V,
    rev vs ++ ws = ws_given ->
      rev2'_v4 V beq_V vs ws_given = Some ws.

```

In words – if applying `rev2'_v4` to `vs` returns a list, this list is a suffix of `ws_given` whose prefix is the reverse of `vs`; and if a list is the reverse prefix of `ws_given`, applying `rev2'_v4` to this list totally yields the corresponding suffix of `ws_given`.

OCaml, however, provides exceptions. Here is a direct-style implementation in OCaml:

```

exception Mismatching_lists;;

let rec rev2'_v4 vs ws_given =
  match vs with
  [] -> ws_given
  | v :: vs' -> (match rev2'_v4 vs' ws_given with
    [] -> raise Mismatching_lists
    | w :: ws' -> if v = w then ws' else raise Mismatching_lists);;

let rev2_v4 vs_given ws_given =
  try match rev2'_v4 vs_given ws_given with
    [] -> true
    | _ :: _ -> false
  with Mismatching_lists -> false;;

```

The codomain of `rev2'_v4` is `bool`, not `bool option`, and an exception is raised in case of mismatch. Otherwise, the successive calls to `rev2'_v4` traverse the first list and its successive returns traverse the second, without creating any intermediate list, which is the hallmark of TABA.

### 3.5. A first-order and recursive implementation, lambda-dropped

As it happens, the implementations of Section 3.4 are recursive equations, i.e., they are in lambda-lifted form (see Appendix C). They can be lambda-dropped into a block-structured, lexically scoped program where the exception is used locally, the auxiliary function is defined locally, and `ws_given` occurs free since it is lexically visible [23]:

```

let rev2_v5 vs_given ws_given = (* 'a list -> 'a list -> bool *)
  try let rec rev2'_v5 vs = (* 'a list -> 'a list *)
      match vs with
      | [] -> ws_given
      | v :: vs' -> (match rev2'_v5 vs' with
                    | [] -> raise Mismatching_lists
                    | w :: ws' -> if v = w then ws' else raise Mismatching_lists)
      in match rev2'_v5 vs_given with
      | [] -> true
      | _ :: _ -> false
    with Mismatching_lists -> false;;

```

Again, this implementation is structurally recursive and therefore it can be expressed using fold-right (see Appendix B). It is also characteristic of TABA: the first list is traversed at call time and the second at return time, and no intermediate data structure is created.

Let us visualize the induced computation with traces:

- In the first example, `rev2_v5` is called with the lists `[1; 2; 3]` and `[3; 2; 1]`, and then calls `rev2'_v5` recursively as it traverses `[1; 2; 3]`. When the end of the list is reached, the second list is returned, and the successive heads of `[1; 2; 3]` that were accessed at call time are compared with the successive heads of `[3; 2; 1]` as they are accessed at return time, until the initial return from `rev2'_v5`. The final result is `true` since all the (traced) tests succeeded and the returned list is empty:

```

# traced_rev2_v5 show_int [1; 2; 3] [3; 2; 1];;
rev2_v5 [1; 2; 3] [3; 2; 1] ->
  rev2'_v5 [1; 2; 3] ->
    rev2'_v5 [2; 3] ->
      rev2'_v5 [3] ->
        rev2'_v5 [] ->
          rev2'_v5 [] <- [3; 2; 1]
          3 = 3 <-> true
          rev2'_v5 [3] <- [2; 1]
          2 = 2 <-> true
          rev2'_v5 [2; 3] <- [1]
          1 = 1 <-> true
          rev2'_v5 [1; 2; 3] <- []
rev2_v5 [1; 2; 3] [3; 2; 1] <- true
- : bool = true
#

```

- In the second example, `rev2_v5` is called with the lists `[1; 2; 3; 4]` and `[4; 0; 2; 1]`, and then calls `rev2'_v5` recursively as it traverses `[1; 2; 3; 4]`. When the end of the list is reached, the second list is returned, and the successive heads of `[1; 2; 3; 4]` that were accessed at call time are compared with the successive heads of `[4; 0; 2; 1]` as they are accessed at return time. The two first heads, 4, match, but the two next heads, 3 and 0 do not. So an exception is raised and the final result is `false`:



```
# traced_rev2_v5 show_int [1; 2; 3; 4] [4; 0; 2; 1];;
rev2_v5 [1; 2; 3; 4] [4; 0; 2; 1] ->
  rev2'_v5 [1; 2; 3; 4] ->
    rev2'_v5 [2; 3; 4] ->
      rev2'_v5 [3; 4] ->
        rev2'_v5 [4] ->
          rev2'_v5 [] ->
            rev2'_v5 [] <- [4; 0; 2; 1]
          4 = 4 <-> true
        rev2'_v5 [4] <- [0; 2; 1]
      3 = 0 <-> false
    rev2_v5 [1; 2; 3; 4] [4; 0; 2; 1] <- false
  - : bool = false
#
```

- The third example illustrates what happens when the second list is shorter than the first, namely the remaining returns are skipped:

```
# traced_rev2_v5 show_int [1; 2] [2];;
rev2_v5 [1; 2] [2] ->
  rev2'_v5 [1; 2] ->
    rev2'_v5 [2] ->
      rev2'_v5 [] ->
        rev2'_v5 [] <- [2]
      2 = 2 <-> true
    rev2'_v5 [2] <- []
  rev2_v5 [1; 2] [2] <- false
  - : bool = false
#
```

- The fourth example illustrates what happens when the second list is longer than the first, namely the final test fails:

```
# traced_rev2_v5 show_int [1] [1; 0];;
rev2_v5 [1] [1; 0] ->
  rev2'_v5 [1] ->
    rev2'_v5 [] ->
      rev2'_v5 [] <- [1; 0]
    1 = 1 <-> true
  rev2'_v5 [1] <- [0]
  rev2_v5 [1] [1; 0] <- false
  - : bool = false
#
```

### 3.6. A more perspicuous solution where both lists are first traversed

In the implementations above, the job of the first pass (the calls) is to get to the end of the first list, and the job of the second pass (the returns) is to traverse the second list, testing whether it is long enough and whether its successive elements coincide with the corresponding successive elements of the first list, and then finally testing whether the second list is actually too long. But there is a simpler algorithm:

1. traverse *both* the given lists in the first pass, to establish whether they have the same length; and then if they do, and only if they do,

2. traverse the second list, only testing whether its successive elements coincide with the corresponding successive elements of the first list, knowing that the two lists have the same length.

Analysis: the number of recursive calls is the same if the two lists have the same length, and otherwise this number is the length of the shorter list; also, the second pass is simpler since it only takes place if the two lists have the same length.

The following sections walk back the path of the previous sections, using this simpler algorithm and assuming that the reader concurs with its tenet.

### 3.5. A more perspicuous first-order and recursive implementation, lambda-dropped

Compared to the implementation in Section 3.5 page 129, the auxiliary function now traverses both of the given lists and raises an exception if they do not have the same length:

```
let rev2_w5 vs_given ws_given =
  try let rec rev2'_w5 vs ws =
      match vs, ws with
      | [], [] -> ws_given
      | [], _ :: _ -> raise Mismatching_lists
      | _ :: _, [] -> raise Mismatching_lists
      | v :: vs', _ :: ws' -> match rev2'_w5 vs' ws' with
          [] -> [] (* impossible case *)
          | x :: xs' -> if v = x then xs' else raise Mismatching_lists
    in let _ = rev2'_w5 vs_given ws_given
        in true
  with Mismatching_lists -> false;;
```

Since `rev2'_w5` only returns if the two given lists have the same length, the nil case in the induction step is impossible, and if the initial call to `rev2'_w5` completes, the result is the empty list.

Let us visualize the induced computation with traces, using the same examples as in Section 3.5 page 129:

- In the first example, `rev2_w5` is called with the lists `[1; 2; 3]` and `[3; 2; 1]`, and then calls `rev2'_w5` recursively as it traverses these two lists. When the end of the two lists is reached, the second list is returned, and the successive heads of `[1; 2; 3]` that were accessed at call time are compared with the successive heads of `[3; 2; 1]` as they are accessed at return time, until the initial return from `rev2'_w5`: the final result is true:

```
# traced_rev2_w5 show_int [1; 2; 3] [3; 2; 1];;
rev2_w5 [1; 2; 3] [3; 2; 1] ->
  rev2'_w5 [1; 2; 3] [3; 2; 1] ->
    rev2'_w5 [2; 3] [2; 1] ->
      rev2'_w5 [3] [1] ->
        rev2'_w5 [] [] ->
          rev2'_w5 [] [] <- [3; 2; 1]
        3 = 3 <-> true
      rev2'_w5 [3] [1] <- [2; 1]
    2 = 2 <-> true
  rev2'_w5 [2; 3] [2; 1] <- [1]
```

```

1 = 1 <-> true
rev2'_w5 [1; 2; 3] [3; 2; 1] <- []
rev2_w5 [1; 2; 3] [3; 2; 1] <- true
- : bool = true
#

```

- In the second example, `rev2_w5` is called with the lists `[1; 2; 3; 4]` and `[4; 0; 2; 1]`, and then calls `rev2'_w5` recursively as it traverses these two lists. When the end of the two lists is reached, the second list is returned, and the successive heads of `[1; 2; 3; 4]` that were accessed at call time are compared with the successive heads of `[4; 0; 2; 1]` as they are accessed at return time. The two first heads, 4, match, but the two next heads, 3 and 0 do not. So an exception is raised and the final result is `false`:

```

# traced_rev2_w5 show_int [1; 2; 3; 4] [4; 0; 2; 1];;
rev2_w5 [1; 2; 3; 4] [4; 0; 2; 1] ->
  rev2'_w5 [1; 2; 3; 4] [4; 0; 2; 1] ->
    rev2'_w5 [2; 3; 4] [0; 2; 1] ->
      rev2'_w5 [3; 4] [2; 1] ->
        rev2'_w5 [4] [1] ->
          rev2'_w5 [] [] ->
            rev2'_w5 [] [] <- [4; 0; 2; 1]
              4 = 4 <-> true
                rev2'_w5 [4] [1] <- [0; 2; 1]
                  3 = 0 <-> false
                    rev2_w5 [1; 2; 3; 4] [4; 0; 2; 1] <- false
                      - : bool = false
                        #

```

- The third example illustrates what happens when the second list is shorter than the first, and is where `rev2_w5` shines compared to `rev2_v5`. The two lists are traversed until the second one reaches the empty list. Then an exception is raised, and the final result is `false`:

```

# traced_rev2_w5 show_int [1; 2] [2];;
rev2_w5 [1; 2] [2] ->
  rev2'_w5 [1; 2] [2] ->
    rev2'_w5 [2] [] ->
      rev2_w5 [1; 2] [2] <- false
        - : bool = false
          #

```

- The fourth example illustrates what happens when the second list is longer than the first, and is also where `rev2_w5` shines compared to `rev2_v5`. The two lists are traversed until the first one reaches the empty list. Then an exception is raised, and the final result is `false`:

```

# traced_rev2_w5 show_int [1] [1; 0];;
rev2_w5 [1] [1; 0] ->
  rev2'_w5 [1] [1; 0] ->
    rev2'_w5 [] [0] ->
      rev2_w5 [1] [1; 0] <- false
        - : bool = false
          #

```

### 3.4. A more perspicuous first-order and recursive implementation, lambda-lifted

Lambda-lifting the implementation of Section 3.5 page 132, yields two recursive equations:

```
let rec rev2'_w4 vs ws ws_given =
  match vs, ws with
  | [], [] -> ws_given
  | [], _ :: _ -> raise Mismatching_lists
  | _ :: _, [] -> raise Mismatching_lists
  | v :: vs', _ :: ws' -> match rev2'_w4 vs' ws' ws_given with
      [] -> [] (* impossible case *)
    | x :: xs' -> if v = x then xs' else raise Mismatching_lists;;

let rev2_w4 vs_given ws_given =
  try let _ = rev2'_w4 vs_given ws_given ws_given in true
  with Mismatching_lists -> false;;
```

The auxiliary function now takes `ws_given` as an extra parameter since it is no longer lexically visible in the nil-nil case.

### 3.3. A more perspicuous higher-order and tail-recursive implementation

CPS-transforming the implementation of Section 3.4 page 134, yields a purely functional program that we can express in Gallina:

```
Fixpoint rev2'_w3 (V : Type) (beq_V : V -> V -> bool)
  (vs ws ws_given : list V) (k : list V -> bool) : bool :=
  match vs, ws with
  | nil, nil => k ws_given
  | nil, _ :: _ => false
  | _ :: _, nil => false
  | v :: vs', _ :: ws' => rev2'_w3 V beq_V vs' ws' ws_given (fun xs =>
      match xs with
      | nil => false
      | x :: xs' => if beq_V v x
        then k xs'
        else false
      end)
  end.

Definition rev2_w3 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  rev2'_w3 V beq_V vs_given ws_given ws_given (fun xs => match xs with
    nil => true
  | _ :: _ => false
  end).
```

In words – `rev2_w3` accumulates the successive elements of the first list into a function `k` to traverse the second list and compare it to the first one. Then, if the two lists have the same length, `k` is applied to the second list to carry out this comparison.

The auxiliary definition above gives rise to two auxiliary lemmas, a corollary of which is the soundness and completeness of `rev2_w3`:

```

Lemma soundness_of_rev2'_w3 :
forall (V : Type) (beq_V : V -> V -> bool),
  (forall v v' : V, beq_V v v' = true -> v = v') ->
  forall (vs ws ws_given : list V) (k : list V -> bool),
    (exists ws_sfx : list V,
      rev2'_w3 V beq_V vs ws ws_given k = k ws_sfx /\ rev vs ++ ws_sfx = ws_given) \/
      rev2'_w3 V beq_V vs ws ws_given k = false.

```

In words – either `rev2'_w3` ends up tail-calling `k` on a suffix of `ws_given` such that the corresponding prefix is the reverse of `vs` or it yields `false`. This soundness lemma is proved by induction on `vs`.

```

Lemma completeness_of_rev2'_w3 :
forall (V : Type) (beq_V : V -> V -> bool),
  (forall v v' : V, v = v' -> beq_V v v' = true) ->
  forall vs ws ws_pfx ws_sfx ws_given : list V,
    length vs = length ws ->
    rev vs = ws_pfx ->
    ws_pfx ++ ws_sfx = ws_given ->
    forall k : list V -> bool,
      rev2'_w3 V beq_V vs ws ws_given k = k ws_sfx.

```

In words – if `vs` is the reverse of a prefix of `ws_given`, then `rev2'_w3` ends up tail-calling `k` on the corresponding suffix if `vs_given` and `ws_given` have the same length. This completeness lemma is proved by induction on `vs`.

### 3.2. A more perspicuous first-order and tail-recursive implementation

The implementation in Section 3.3 page 134, is a candidate for defunctionalization, using lists as a data type for the continuation and `rev2''_w2` as the corresponding apply function, which is only called if its arguments have the same length and which tests their structural equality:

```

Fixpoint rev2''_w2 (V : Type) (beq_V : V -> V -> bool) (vs_op xs : list V) : bool :=
  match vs_op, xs with
  | nil, nil => true
  | nil, _ :: _ => false
  | _ :: _, nil => false
  | v :: vs'_op, x :: xs' => if beq_V v x
    then rev2''_w2 V beq_V vs'_op xs'
    else false
  end.

```

```

Fixpoint rev2'_w2 (V : Type) (beq_V : V -> V -> bool) (vs vs_op ws ws_given : list V) :=
  match vs, ws with
  | nil, nil => rev2''_w2 V beq_V vs_op ws_given
  | nil, w :: ws' => false
  | v :: vs', nil => false
  | v :: vs', w :: ws' => rev2''_w2 V beq_V vs' (v :: vs_op) ws' ws_given
  end.

```

```

Definition rev2_w2 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  rev2'_w2 V beq_V vs_given nil ws_given ws_given.

```

Soundness and completeness are proved by structural induction, capturing that the continuation, which has been defunctionalized into `vs_op` and `rev2''_w2`, did implement a comparison between `vs_op` and its argument:

```
Lemma soundness_and_completeness_of_rev2''_w2 :
  forall (V : Type) (beq_V : V -> V -> bool),
    (forall v v' : V, beq_V v v' = true <-> v = v') ->
      forall vs_op ws : list V,
        rev2''_w2 V beq_V vs_op ws = true <-> vs_op = ws.
```

In words – `rev2''_w2` is a sound and complete implementation of structural equality.

We are now in position to prove a lemma about `rev2'_w2`, a corollary of which is the soundness and completeness of `rev2_w2`:

```
Lemma about_rev2'_w2 :
  forall (V : Type) (beq_V : V -> V -> bool) (vs vs_op ws ws_given : list V),
    (exists vs_given_op : list V,
      rev2'_w2 V beq_V vs vs_op ws ws_given = rev2''_w2 V beq_V vs_given_op ws_given /\
      rev vs ++ vs_op = vs_given_op /\ length vs = length ws) \/
    (rev2'_w2 V beq_V vs vs_op ws ws_given = false /\ length vs <> length ws).
```

In words – either `rev2'_w2` ends up tail-calling `rev2''_w2` properly and the two given lists have the same length or it does not and they do not. This lemma is proved by induction on `vs`.

### 3.1. A more perspicuous first-order implementation in two passes

The implementation in Section 3.2 page 135, is a candidate for lightweight fission by fixed-point demotion (see Appendix D) since `rev2'_w2` is tail recursive:

```
Fixpoint rev2'_w1 (V : Type) (vs vs_op ws : list V) : option (list V) :=
  match vs, ws with
  | nil, nil => Some vs_op
  | nil, w :: ws' => None
  | v :: vs', nil => None
  | v :: vs', w :: ws' => rev2'_w1 V vs' (v :: vs_op) ws'
  end.
```

```
Definition rev2_w1 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  match rev2'_w1 V vs_given nil ws_given with
  | Some vs_op => rev2''_w2 V beq_V vs_op ws_given
  | None => false
  end.
```

In words:

1. Both the first list and the second list are traversed, using `rev2'_w1`, and a reversed copy of the first list is returned if the two lists have the same length.
2. If the two lists have the same length, the reversed first list and the second list are traversed together, tail recursively, using `rev2''_w2`. The result is a Boolean: either these two lists have the same elements, i.e., are structurally equal, or they do not.

We can now prove a lemma about `rev2'_w1`, a corollary of which is the soundness and completeness of `rev2_w1`:

```
Lemma about_rev2'_w1 :
  forall (V : Type) (vs vs_op ws : list V),
    (exists vs_given_op : list V,
      rev2'_w1 V vs vs_op ws = Some vs_given_op /\
      rev vs ++ vs_op = vs_given_op /\ length vs = length ws) \/
    (rev2'_w1 V vs vs_op ws = None /\ length vs <> length ws).
```

In words – either `rev2'_w1` yields the reverse of the first given list and the two given lists have the same length, or it does not and they do not. This lemma is proved by induction on `vs`.

### 3.0. Summary, synthesis, and significance

There are many ways to implement a function testing whether two given lists are reverses of each other. Each of these ways reflects a particular vision of this computation: tail recursive with an intermediate list, tail recursive with a continuation, in direct style with an exception, or using a fold function. These implementations form a spectrum in that they can be inter-derived and thus they all reflect the TABA programming pattern, even the non-solutions that allocate intermediate lists. Being structurally recursive, they can be reasoned about equationally using structural induction.

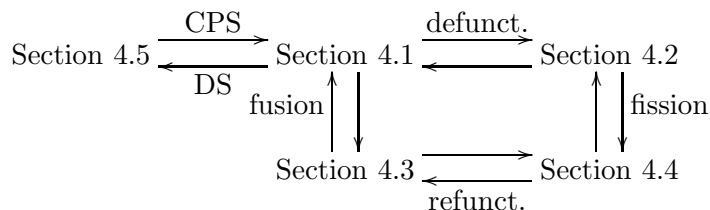
## 4. Convolver two lists

The goal of this section is to implement a function that symbolically convolves two given lists of unknown length when these two lists have the same length, which is the original motivation for TABA [24]. As in Section 3, we are going to inter-derive a spectrum of implementations of a polymorphic function `cnv` that satisfies the following theorem:

```
Theorem soundness_and_completeness_of_cnv :
  forall (V W : Type) (vs : list V) (ws : list W) (ps : list (V * W)),
    cnv V W vs ws = Some ps <-> map fst ps = vs /\ map snd ps = rev ws.
```

For brevity, these implementations are written in OCaml below, with a polymorphic type that reads `'a list -> 'b list -> ('a * 'b) list option`.

We successively consider implementations in continuation-passing style (Section 4.1), their defunctionalized counterparts (Section 4.2) and their lightweight-fissioned counterpart (Section 4.3), and then the version after both defunctionalization and lightweight fusion in either order (Section 4.4). We then consider what it takes—i.e., which control operators one needs—to express the implementations in direct style (Section 4.5). Diagrammatically (each of the transformations is reversible):



We then turn to the more perspicuous analogue of Section 3.6, i.e., traversing both given lists at call time to determine there and then whether the two lists have the same length, thus ensuring that there is only a return time when the two lists have the same length (Section 4.6). And from then on, i.e., from Section 4.7 to Section 4.11, we consider more perspicuous implementations in continuation-passing style, their refunctionalized or/and lightweight-fissioned counterparts, and what it takes to express the more perspicuous implementations in direct style, giving rise to a similar diagram. Section 4.12 draws lessons from these inter-derivations.

## 4.1. Implementations in continuation-passing style

This section presents two implementations of the convolution function using continuations:

1. one implementation first traverses the first list and then returns over the second list; during this return, the resulting list of pairs is accumulated tail-recursively (the call to `k` is a tail call);
2. the other implementation first traverses the second list and then returns over the first list; during this return, the resulting list of pairs is constructed recursively (the call to `k` is not a tail call).

### 4.1.1. Version that returns over the second list

The first given list is traversed tail-recursively with `walk`, a continuation is accumulated, and eventually this continuation is applied to the second list and to an empty list of pairs, yielding an optional list of pairs. The second list is then traversed tail-recursively and a list of pairs is accumulated. If the two given lists have the same length, this list of pairs is returned as the result in the initial continuation; if the second list is shorter than the first, the computation is discontinued, i.e., the continuation is not applied and the result is `None`; and if the second list is longer than the first, the initial continuation returns `None`:

```
let cnv1_cb xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs k = (* 'a list -> ('b list -> ('a * 'b) list option) ->
                      ('a * 'b) list option *)
    match xs with
    | []      -> k ys_given []
    | x :: xs' -> walk xs' (fun ys ps -> match ys with
                                         | []      -> None
                                         | y :: ys' -> k ys' ((x, y) :: ps))
  in walk xs_given (fun ys ps -> match ys with
                               | []      -> Some ps
                               | _ :: _ -> None);;
```

This implementation is the motivation for TABA and is due to Goldberg [24]. It is structurally recursive and therefore can be expressed using `fold-right` and reasoned about using structural induction, since it is also pure (i.e., uses no computational effects).

### 4.1.2. Version that returns over the first list

The second given list is traversed tail-recursively with `walk`, a continuation is accumulated, and eventually this continuation is applied to the first list, yielding a list of pairs. The first list is then traversed



recursively and a list of pairs is constructed. If the two given lists have the same length, this list of pairs is returned as the result; if the first list is shorter than the second, the computation is discontinued, i.e., an exception is raised and the result is `None`; and if the first list is longer than the second, an exception is raised in the initial continuation and the result is `None`:

```
let cnv2_cb xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  try let rec walk ys k = (* 'b list -> ('a list -> ('a * 'b) list) ->
                           ('a * 'b) list option *)
      match ys with
      []       -> k xs_given
    | y :: ys' -> walk ys' (fun xs -> match xs with
                                     []       -> raise Mismatching_lists
                                   | x :: xs' -> (x, y) :: k xs')
    in Some (walk ys_given (fun xs -> match xs with
                                     []       -> []
                                   | _ :: _ -> raise Mismatching_lists))
  with Mismatching_lists -> None;;
```

This implementation is structurally recursive and therefore it can be expressed using fold-right. It is however impure due to the exception that is used to handle the case where the first list does not have the same length as the second. To make it pure, one should change the codomain of the continuation, e.g., using an option type.

## 4.2. First-order (defunctionalized) implementations

This section is dedicated to the counterparts of the implementations of Section 4.1 after defunctionalization, where the defunctionalized continuation is represented as an intermediate list (named either `xs_op` or `ys_op` instead of `k`), together with a second pass (named `continue`) that consumes this intermediate list.

### 4.2.1. Version that returns over the second list

The first given list is traversed tail-recursively with `walk`, a list is accumulated in reverse order, and eventually this reversed list and the second list are traversed in parallel with `continue`, which traverses the two lists tail-recursively and accumulates a list of pairs. If the two given lists have the same length, `continue` returns this list of pairs as the result:

```
let cnv1_fo xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs xs_op = (* 'a list -> 'a list -> ('a * 'b) list option *)
      match xs with
      []       -> continue xs_op ys_given []
    | x :: xs' -> walk xs' (x :: xs_op)
  and continue xs_op ys ps = (* 'a list -> 'b list -> ('a * 'b) list option ->
                              ('a * 'b) list option *)
      match xs_op with
      [] -> (match ys with
              [] -> Some ps
            | _ :: _ -> None)
```



This implementation is structurally recursive and it also fits the pattern of fold-left (see Appendix B):

```
let cnv1_cb_lfi_left xs_given ys_given =
  list_fold_left (fun ys ps -> match ys with
    [] -> Some ps
    | _ :: _ -> None)
    (fun x k ys ps -> match ys with
      [] -> None
      | y :: ys' -> k ys' ((x, y) :: ps))
    xs_given
    ys_given
  [];;
```

We note that substituting fold-right for fold-left in this implementation makes the resulting function implement a dot-product in reverse order, a consequence of constructing the resulting list of pairs tail-recursively using an accumulator. (So did substituting fold-left for fold-right in Section 4.1.1 for the same reason.)

### 4.3.2. Version that returns over the first list

The second given list is traversed tail-recursively with `walk`, a continuation is accumulated, and eventually this continuation is returned. This continuation is then applied to the first list, the first list is traversed recursively, and a list of pairs is constructed, as in Section 4.1.2:

```
let cnv2_cb_lfi xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  try let rec walk ys k = (* 'b list -> ('a list -> ('a * 'b) list) ->
    'a list -> ('a * 'b) list *)
    match ys with
    [] -> k
    | y :: ys' -> walk ys' (fun xs -> match xs with
      [] -> raise Mismatching_lists
      | x :: xs' -> (x, y) :: k xs')
    in Some (walk ys_given (fun xs -> match xs with
      [] -> []
      | _ :: _ -> raise Mismatching_lists) xs_given)
  with Mismatching_lists -> None;;
```

This implementation is structurally recursive and it also fits the pattern of fold-left:

```
let cnv2_cb_lfi_left xs_given ys_given =
  try Some (list_fold_left (fun xs -> match xs with
    [] -> []
    | _ :: _ -> raise Mismatching_lists)
    (fun y k xs -> match xs with
      [] -> raise Mismatching_lists
      | x :: xs' -> (x, y) :: k xs')
    ys_given
    xs_given)
  with Mismatching_lists -> None;;
```

We note that substituting fold-right for fold-left in this implementation makes the resulting function implement a dot-product, a consequence of constructing the resulting list of pairs recursively. (So did substituting fold-left for fold-right in Section 4.1.2 for the same reason.)

#### 4.4. First-order (defunctionalized) implementations after lightweight fission

This section is dedicated to the counterparts of the implementations of Section 4.2 after lightweight fusion, which are also the counterparts of the implementations of Section 4.3 after defunctionalization. They construct an intermediate list as the reverse of one of the two given lists, and then traverse this reversed list and the other given list.

##### 4.4.1. Version that returns over the second list

The first given list is traversed tail-recursively with `walk`, a list is accumulated in reverse order, and eventually this list is returned. This reversed list and the second list are traversed tail-recursively and in parallel with `continue`, and a list of pairs is accumulated. If the two given lists have the same length, this list of pairs is returned as the result:

```
let cnv1_fo_lfi xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs xs_op = (* 'a list -> 'a list -> 'a list *)
    match xs with
    | [] -> xs_op
    | x :: xs' -> walk xs' (x :: xs_op)
  and continue xs_op ys ps = (* 'a list -> 'b list -> ('a * 'b) list option *)
    match xs_op with
    | [] -> (match ys with
              | [] -> Some ps
              | _ :: _ -> None)
    | x :: xs'_op -> (match ys with
                     | [] -> None
                     | y :: ys' -> continue xs'_op ys' ((x, y) :: ps))
  in continue (walk xs_given []) ys_given [];;
```

The definition of `walk` coincides with the tail-recursive definition of `reverse` that uses an accumulator, and `continue` tail-recursively zips together the two lists it is applied to, using an accumulator.

##### 4.4.2. Version that returns over the first list

The second given list is traversed tail-recursively with `walk`, a list is accumulated in reverse order, and eventually this list is returned. The first list and this reversed list are traversed recursively and in parallel with `continue`, and a list of pairs is constructed. If the two given lists have the same length, this list of pairs is returned as the result:

```
let cnv2_fo_lfi xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  try let rec walk ys ys_op = (* 'b list -> 'b list -> 'b list *)
    match ys with
    | [] -> ys_op
    | y :: ys' -> walk ys' (y :: ys_op)
  and continue ys_op xs = (* 'b list -> 'a list -> ('a * 'b) list *)
    match ys_op with
    | [] -> (match xs with
              | [] -> []
              | _ :: _ -> raise Mismatching_lists)
```

```

| y :: ys'_op -> (match xs with
                 []      -> raise Mismatching_lists
                 | x :: xs' -> (x, y) :: continue ys'_op xs')
in Some (continue (walk ys_given []) xs_given)
with Mismatching_lists -> None;;

```

The definition of `walk` coincides with the tail-recursive definition of `reverse` that uses an accumulator, and `continue` recursively zips together the two lists it is applied to.

## 4.5. Towards implementations in direct style

This section is dedicated to the direct-style counterpart of the continuation-passing implementations in Section 4.1. The first one is straightforward (and uses an exception), and the second is not (it involves delimited-control operators).

### 4.5.1. First version: reversing the first list

Since `cnv1_cb`, in Section 4.1.1, is tail recursive throughout, it is simple to express it in direct style, using an exception to handle the case where the current continuation is not applied:

```

let cnv1_dse xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  try let rec walk xs = (* 'a list -> 'b list * ('a * 'b) list *)
      match xs with
      []      -> (ys_given, [])
      | x :: xs' -> let (ys, ps) = walk xs'
                    in match ys with
                       []      -> raise Mismatching_lists
                       | y :: ys' -> (ys', (x, y) :: ps)
    in let (ys, ps) = walk xs_given
        in match ys with
           []      -> Some ps
           | _ :: _ -> None
    with Mismatching_lists -> None;;

```

### 4.5.2. Second version: reversing the second list

In Section 4.1.2, the call to the continuation is not a tail call in the induction step. The definition of `cnv2_cb` therefore provides yet another case for the delimited-control operators `shift` and `reset` [25], and yet another illustration of the type mismatch between the codomain of the continuation that has no option, and the domain of answers that has an option [26, 27]. We choose to let this sleeping dog lie, as awakening it would require more background material in a way that is unrelated to TABA.

## 4.6. A more perspicuous solution where both lists are first traversed

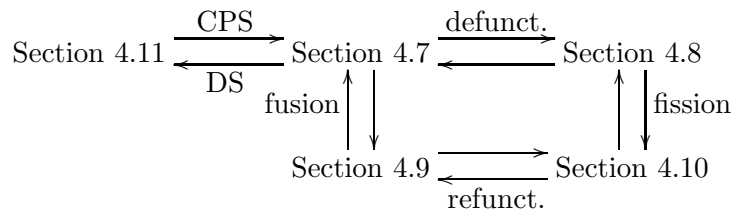
In the implementations above, the job of the first pass (the calls) is to get to the end of one of the given lists, and the job of the second pass (the returns) is to traverse the other given list, checking that it is long enough, pairing the successive elements of the first list at the point of call together with the

successive elements of the second list at the corresponding point of return, grouping these pairs into a list, and then finally testing whether the other list is actually too long. But as in Section 3.6 there is a simpler algorithm:

1. traverse *both* the given lists in the first pass, to establish whether they have the same length; and then if they do, and only if they do,
2. traverse the other list to construct the list of pairs, knowing that the two given lists have the same length.

Analysis: as in Section 3.6, the number of recursive calls is the same if the two lists have the same length, and otherwise this number is the length of the shorter list; also, the second pass is simpler since it only takes place if the two lists have the same length.

The following sections walk through the path of the previous sections, using this simpler algorithm:



## 4.7. More perspicuous implementations in continuation-passing style

This section is dedicated to two implementations of the convolution function using continuations. Each of these implementations traverses both lists tail recursively and in parallel. If these lists do not have the same length, the result is `None`. Otherwise,

1. the first implementation returns over the second list; during this return, the resulting list of pairs is accumulated tail-recursively (the call to `k` is a tail call);
2. the second implementation returns over the first list; during this return, the resulting list of pairs is constructed recursively (the call to `k` is not a tail call).

### 4.7.1. Version that returns over the second list

Both given lists are traversed tail-recursively with `walk`, as a continuation is accumulated. If the two lists do not have the same length, the continuation is ignored and the (optional) result is nothing. Otherwise, the continuation is applied to the second list and to an empty list of pairs. The second list is then traversed tail-recursively and a list of pairs is accumulated. The (optional) result is the final version of the accumulator:

```

let cnw1_cb xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys k = (* 'a list -> 'b list ->
                          ('b list -> ('a * 'b) list -> ('a * 'b) list) ->
                          ('a * 'b) list option *)
    match xs, ys with
    | [], [] -> Some (k ys_given [])
    | x :: xs', _ :: ys' -> walk xs' ys' (fun ys ps ->
                                          match ys with
                                          | [] -> [] (* impossible case *)
                                          | y :: ys' -> k ys' ((x, y) :: ps))
    | _, _ -> None
  in walk xs_given ys_given (fun _ ps -> ps);;

```

#### 4.7.2. Version that returns over the first list

Both given lists are traversed tail-recursively with `walk` as a continuation is accumulated. If the two lists do not have the same length, the continuation is ignored and the (optional) result is nothing. Otherwise, the continuation is applied to the first list. The first list is then traversed recursively and a list of pairs is constructed. The (optional) result is this list of pairs:

```

let cnw2_cb xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys k = (* 'a list -> 'b list -> ('a list -> ('a * 'b) list) ->
                          ('a * 'b) list option *)
    match xs, ys with
    | [], [] -> Some (k xs_given)
    | _ :: xs', y :: ys' -> walk xs' ys' (fun xs ->
                                          match xs with
                                          | [] -> [] (* impossible case *)
                                          | x :: xs' -> (x, y) :: k xs')
    | _, _ -> None
  in walk xs_given ys_given (fun _ -> []);;

```

The nil case in the continuation is impossible, so compared to `cnv2_cb` in Section 4.1.2, there is no need for an exception.

## 4.8. More perspicuous first-order (defunctionalized) implementations

This section is dedicated to the counterparts of the implementations of Section 4.7 after defunctionalization, where the defunctionalized continuation is represented as an intermediate list (named either `xs_op` or `ys_op` instead of `k`), together with the second pass (named `continue`) that consumes this intermediate list.

### 4.8.1. Version that returns over the second list

Both given lists are traversed tail-recursively with `walk` as the reverse of the first list is accumulated. If the two lists do not have the same length, the result is `None`. Otherwise, the reversed first list and the second list are zipped together tail recursively with `continue`, using an accumulator, and the (optional) result is the final value of this accumulator:

```

let cnw1_fo xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys xs_op = (* 'a list -> 'b list -> 'a list -> ('a * 'b) list option *)
    match xs, ys with
    | [], [] -> Some (continue xs_op ys_given [])
    | x :: xs', _ :: ys' -> walk xs' ys' (x :: xs_op)
    | _, _ -> None
  and continue xs_op ys ps = (* 'a list -> 'b list -> ('a * 'b) list -> ('a * 'b) list *)
    match xs_op with
    | [] -> ps
    | x :: xs'_op -> (match ys with
      | [] -> [] (* impossible case *)
      | y :: ys' -> continue xs'_op ys' ((x, y) :: ps))
  in walk xs_given ys_given [];;

```

#### 4.8.2. Version that returns over the first list

Both given lists are traversed tail-recursively with `walk` as the reverse of the second list is accumulated. If the two lists do not have the same length, the result is `None`. Otherwise, the first list and the reversed second list are zipped together recursively with `continue`, and the (optional) result is the resulting list of pairs:

```

let cnw2_fo xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys ys_op = (* 'a list -> 'b list -> 'b list -> ('a * 'b) list option *)
    match xs, ys with
    | [], [] -> Some (continue ys_op xs_given)
    | _ :: xs', y :: ys' -> walk xs' ys' (y :: ys_op)
    | _, _ -> None
  and continue ys_op xs = (* 'b list -> 'a list -> ('a * 'b) list *)
    match ys_op with
    | [] -> []
    | y :: ys'_op -> match xs with
      | [] -> [] (* impossible case *)
      | x :: xs' -> (x, y) :: continue ys'_op xs'
  in walk xs_given ys_given [];;

```

The `nil` case in `continue` is impossible, so compared to `cnw2_fo` in Section 4.2.2, there is no need for an exception.

### 4.9. More perspicuous implementations in continuation-passing style after lightweight fission

This section is dedicated to the counterparts of the implementations of Section 4.7 after lightweight fission. Their key point is that the auxiliary function returns the continuation instead of applying it.

#### 4.9.1. Version that returns over the second list

Both given lists are traversed tail-recursively with `walk`, as a continuation is accumulated. If the two lists do not have the same length, the continuation is ignored and the result is `None`. Otherwise, the continuation is returned, and is then applied to the second list and an empty list of pairs; the second



list is traversed tail-recursively, and a list of pairs is accumulated and eventually returned as the result in the initial continuation:

```
let cnw1_cb_lfi xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys k = (* 'a list -> 'b list ->
                        ('b list -> ('a * 'b) list -> ('a * 'b) list) option *)
    match xs, ys with
    | [], [] -> Some k
  | x :: xs', _ :: ys' -> walk xs' ys' (fun ys ps ->
                                        match ys with
                                        | [] -> [] (* impossible case *)
                                        | y :: ys' -> k ys' ((x, y) :: ps))
  | _, _ -> None
  in match walk xs_given ys_given (fun _ ps -> ps) with
  | Some k -> Some (k ys_given [])
  | None -> None ;;
```

#### 4.9.2. Version that returns over the first list

Both given lists are traversed tail-recursively with walk as a continuation is accumulated. If the two lists do not have the same length, the continuation is ignored and the result is None. Otherwise, the continuation is applied to the first list, which is traversed recursively, and a list of pairs is constructed that forms the (optional) result:

```
let cnw2_cb_lfi xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys k = (* 'a list -> 'b list -> ('a list -> ('a * 'b) list)) option *)
    match xs, ys with
    | [], [] -> Some k
  | _ :: xs', y :: ys' -> walk xs' ys' (fun xs ->
                                        match xs with
                                        | [] -> [] (* impossible case *)
                                        | x :: xs' -> (x, y) :: k xs')
  | _, _ -> None
  in match walk xs_given ys_given (fun _ -> []) with
  | Some k -> Some (k xs_given)
  | None -> None ;;
```

The nil case in the continuation is impossible, so compared to `cnw2_cb_lfi` in Section 4.3.2, there is no need for an exception.

### 4.10. More perspicuous first-order (defunctionalized) implementations after lightweight fission

This section is dedicated to the counterparts of the implementations of Section 4.9 after defunctionalization, where the defunctionalized continuation is represented as an intermediate list (named either `xs_op` or `ys_op` instead of `k`), together with the second pass (named `continue`) that consumes this intermediate list. Their key point is that they are non-solutions since they construct an intermediate list as the reverse of one of the two given lists, and then traverse this reversed list and the other given list.

#### 4.10.1. Version that returns over the second list

Both given lists are traversed tail-recursively with `walk` as the reverse of the first list is accumulated. If the two lists do not have the same length, the result is `None`. Otherwise, the reversed first list is returned. The reversed first list and the second list are then zipped together tail recursively with `continue`, using an accumulator, and the (optional) result is the final version of this accumulator:

```
let cnw1_fo_lfi xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys xs_op = (* 'a list -> 'b list -> 'a list -> 'a list option *)
    match xs, ys with
    | [], [] -> Some xs_op
    | x :: xs', _ :: ys' -> walk xs' ys' (x :: xs_op)
    | _, _ -> None
  and continue xs_op ys ps = (* 'a list -> 'b list -> ('a * 'b) list -> ('a * 'b) list *)
    match xs_op with
    | [] -> ps
    | x :: xs'_op -> (match ys with
      | [] -> [] (* impossible case *)
      | y :: ys' -> continue xs'_op ys' ((x, y) :: ps))
  in match walk xs_given ys_given [] with
    Some xs_op -> Some (continue xs_op ys_given [])
    | None -> None;;
```

#### 4.10.2. Version that returns over the first list

Both given lists are traversed tail-recursively with `walk` as the reverse of the second list is accumulated. If the two lists do not have the same length, the result is `None`. Otherwise, the reversed second list is returned. The first list and the reversed second list are then zipped together recursively with `continue`, and the (optional) result is the resulting list of pairs:

```
let cnw2_fo_lfi xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  let rec walk xs ys ys_op = (* 'a list -> 'b list -> 'b list -> 'b list option *)
    match xs, ys with
    | [], [] -> Some ys_op
    | _ :: xs', y :: ys' -> walk xs' ys' (y :: ys_op)
    | _, _ -> None
  and continue ys_op xs = (* 'b list -> 'a list -> ('a * 'b) list *)
    match ys_op with
    | [] -> []
    | y :: ys'_op -> match xs with
      | [] -> [] (* impossible case *)
      | x :: xs' -> (x, y) :: continue ys'_op xs'
  in match walk xs_given ys_given [] with
    Some ys_op -> Some (continue ys_op xs_given)
    | None -> None;;
```

The `nil` case in `continue` is impossible, so compared to `cnv2_fo_lfi` in Section 4.4.2, there is no need for an exception.

## 4.11. Towards more perspicuous implementations in direct style

This section is dedicated to the direct-style counterpart of the continuation-passing implementations in Section 4.7. The first one is straightforward (and uses an exception), and the second is not (it involves delimited-control operators).

### 4.11.1. Version that returns over the second list

As in Section 4.5.1, it is simple to express `cnw1_cb` in direct style, using an exception:

```
let cnw1_dse xs_given ys_given = (* 'a list -> 'b list -> ('a * 'b) list option *)
  try let rec walk xs ys = (* 'a list -> 'b list -> 'b list * ('a * 'b) list *)
      match xs, ys with
      | [], [] -> (ys_given, [])
      | x :: xs', _ :: ys' -> let (ys, ps) = walk xs' ys'
          in (match ys with
              | [] -> (ys, ps) (* impossible case *)
              | y :: ys' -> (ys', (x, y) :: ps))
          in let (_, ps) = walk xs_given ys_given in Some ps
        with Mismatching_lists -> None;;
```

### 4.11.2. Version that returns over the first list

As in Section 4.5.2, the definition of `cnw2_cb` provides yet another case for shift and reset, which again we refrain from elaborating.

## 4.12. Summary, synthesis, and significance

Throughout, each two versions contrasts the tail-recursive accumulation of the resulting list of pairs and the recursive construction of this resulting list. The spectrum of implementations ranges from completely explicit (two passes and an intermediate (reversed) list) to completely implicit (one recursive descent that is possibly interrupted by raising an exception). That lightweight fission makes a continuation-passing implementation expressible using fold-left came as a surprise to the author, and seems new. The effect of replacing one fold functional by the other crystallizes the relation between convolving two lists and constructing their dot-product: a recursive construction yields the resulting list of pairs in the order of the two given lists, and a tail-recursive accumulation yields the resulting list of pairs in the reverse order.

As for reasoning about these implementations (the pure ones, that is), it involves the same apparatus as in Section 3: structural induction and equational reasoning.

## 5. Deciding whether a lambda term has the shape of an eta redex

Given a  $\lambda$  term, we want to know whether it has the shape  $\lambda x_1. \lambda x_2. \dots \lambda x_n. e x_1 x_2 \dots x_n$ , for some (unknown) expression  $e$  and depth  $n$ . If it does, we would like to know this expression and this depth, in  $n$  recursive calls. We refer to a term of this shape as “an  $\eta$  redex,” a slight abuse of terminology

since  $\lambda x.x x$  is not an  $\eta$  redex, for example, but this abuse is inessential here. We first consider  $\lambda$  terms with names to determine whether they have the shape of  $\lambda x_1.\lambda x_2.\dots\lambda x_n.e x_1 x_2 \dots x_n$  for some  $e$  and  $n$  (Section 5.1, using OCaml), then  $\lambda$  terms with de Bruijn levels [28] to determine whether they have the shape of  $\lambda \lambda \dots \lambda e 0 1 \dots (n-1)$  for some  $e$  and  $n$  (Section 5.2, using the Coq Proof Assistant), and then  $\lambda$  terms with de Bruijn indices [28] to determine whether they have the shape of  $\lambda \lambda \dots \lambda e (n-1) (n-2) \dots 0$  for some  $e$  and  $n$  (Section 5.3, using the Coq Proof Assistant).

## 5.1. Lambda terms with names

Here is a stylized data type for the abstract-syntax trees of  $\lambda$  terms with names, using a unary constructor `Expn` to stand for any term that is not a variable, a  $\lambda$  abstraction, or an application:

```
type expn = Varn of string | Lamn of string * expn | Appn of expn * expn
          | Expn of string;;
```

An  $\eta$  redex of depth  $d$  reads  $\lambda x_1.\lambda x_2.\dots\lambda x_d.e x_1 x_2 \dots x_d$ , but in actuality, this redex contains many implicit parentheses and actually reads  $\lambda x_1.(\lambda x_2.\dots(\lambda x_d.((\dots((e x_1) x_2) \dots) x_d)) \dots)$ . Reflecting these implicit parentheses, the following function constructs an  $\eta$  redex, given a term and a depth:

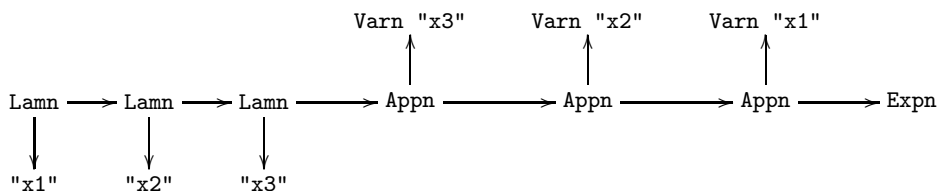
```
let make_eta_redexn e d = (* expn -> int -> expn *)
  assert (d > 0);
  let rec visit e d c = (* expn -> int -> int -> expn *)
    if d = 0
    then e
    else let x = "x" ^ string_of_int c
          in Lamn (x, visit (Appn (e, Varn x)) (pred d) (succ c))
  in visit e d 1;;
```

In words – `visit` recursively constructs the resulting term with `Lamn` while accumulating the body around the given expression with `Appn`. Characteristically of  $\lambda$  terms with names, a (hopefully) fresh name is generated for each instance of `Lamn` using a counter that is initialized with 1.

For example, evaluating `make_eta_redexn Exp 3` yields

```
Lamn ("x1",
  Lamn ("x2",
    Lamn ("x3",
      Appn (Appn (Appn (Expn "x0",
                    Varn "x1"),
                  Varn "x2"),
            Varn "x3"))))
```

Here is its abstract-syntax tree, rotated 90 degrees counterclockwise and flattened:



Detecting whether a given  $\lambda$  term with names has the shape of an  $\eta$  redex fits TABA in that we can recursively descend into its abstract-syntax tree for as long as we encounter `Lamn` constructors, until we encounter another constructor. At that point, if the tree has the shape of an  $\eta$  redex, its depth is the number of recursive calls that have taken us there. We can then return the tree and keep traversing it at return time for as long as we encounter `Appn` constructors whose second arguments are a variable whose name matches the name in the corresponding `Lamn` constructor. (This name is visible in the lexical environment of the current recursive call.) If any other constructor than `Lamn` (during the calls) and `Appn` (during the returns) is encountered, the term is not an  $\eta$  redex. The term is an  $\eta$  redex if the calls and the returns encounter as many occurrences of `Lamn` and of `Appn` and if the second argument of each occurrence of `Appn` (i.e., the actual parameter) is a variable whose name matches the name declared in the matching `Lamn` constructor (i.e., the formal parameter).

The following implementation is  $\lambda$  dropped and uses TABA in direct style with an option through-out:

```
let etapn_dso e = (* expn -> (expn * int) option *)
  let rec walk e = (* expn -> (expn * int) option *)
    match e with
    | Lamn (x, e') -> (match walk e' with
      | Some (Appn (b, Varn x'), d) -> if x' = x
        then Some (b, succ d)
        else None
      | _ -> None)
    | _ -> Some (e, 0)
  in match walk e with
  | Some (b, d) -> if d = 0 then None else Some (b, d)
  | None -> None;
```

where `etapn_dso` stands for “eta predicate for  $\lambda$  terms with names in direct style using an option type.” At the outset, we check that the depth of the putative  $\eta$  redex is not 0, i.e., is strictly positive.

Let us visualize the induced computation with a trace, rendering the  $\lambda$  terms using the syntax of Scheme:

```
# traced_etapn_dso (make_eta_redexn (Expn "x0") 3);;
etapn_dso (lambda (x1) (lambda (x2) (lambda (x3) (((x0 x1) x2) x3)))) ->
walk (lambda (x1) (lambda (x2) (lambda (x3) (((x0 x1) x2) x3)))) ->
walk (lambda (x2) (lambda (x3) (((x0 x1) x2) x3))) ->
walk (lambda (x3) (((x0 x1) x2) x3)) ->
walk (((x0 x1) x2) x3) ->
walk (((x0 x1) x2) x3) <- Some (((x0 x1) x2) x3), 0)
x3 = x3 <-> true
walk (lambda (x3) (((x0 x1) x2) x3)) <- Some (((x0 x1) x2), 1)
x2 = x2 <-> true
walk (lambda (x2) (lambda (x3) (((x0 x1) x2) x3))) <- Some ((x0 x1), 2)
x1 = x1 <-> true
walk (lambda (x1) (lambda (x2) (lambda (x3) (((x0 x1) x2) x3)))) <- Some (x0, 3)
etapn_dso (lambda (x1) (lambda (x2) (lambda (x3) (((x0 x1) x2) x3)))) <- Some (x0, 3)
- : (expn * int) option = Some (Expn "x0", 3)
#
```

The successive Lamn constructors are traversed through the calls to walk and the successive Appn constructors are traversed through the subsequent returns, since the successive names match.

Of course most terms are not an  $\eta$  redex. Then the result is None, as illustrated below with Curry's B combinator:

```
# traced_etapn_dso
  (Lamn ("x",
        Lamn ("y",
              Lamn ("z",
                    Appn (Appn (Expn "I", Varn "x"), Appn (Varn "y", Varn "z")))))));;
etapn_dso (lambda (x) (lambda (y) (lambda (z) ((I x) (y z))))) ->
  walk (lambda (x) (lambda (y) (lambda (z) ((I x) (y z))))) ->
  walk (lambda (y) (lambda (z) ((I x) (y z))))) ->
  walk (lambda (z) ((I x) (y z))) ->
  walk ((I x) (y z)) ->
  walk ((I x) (y z)) <- Some (((I x) (y z)), 0)
  walk (lambda (z) ((I x) (y z))) <- None
  walk (lambda (y) (lambda (z) ((I x) (y z)))) <- None
  walk (lambda (x) (lambda (y) (lambda (z) ((I x) (y z))))) <- None
etapn_dso (lambda (x) (lambda (y) (lambda (z) ((I x) (y z))))) <- None
- : (expn * int) option = None
#
```

To short-circuit the intermediate returns of None if the given term is not an  $\eta$  redex, we can also use an exception:

```
let etapn_dse e = (* expn -> (expn * int) option *)
  try let rec walk e = (* expn -> expn * int *)
    match e with
      Lamn (x, e') -> (match walk e' with
        (Appn (b, Varn x'), d) -> if x' = x
          then (b, succ d)
          else raise Not_a_redex
        | _ -> raise Not_a_redex)
      | _ -> (e, 0)
    in let (b, d) = walk e
      in if d = 0 then None else Some (b, d)
  with Not_a_redex -> None;;
```

The codomain of walk is now alleviated to be  $\text{expn} * \text{int}$  instead of  $(\text{expn} * \text{int}) \text{option}$  since this function only returns if the given term has not proved yet not to be an  $\eta$  redex. The short circuit can be illustrated with Curry's C combinator:

```
# traced_etapn_dse
  (Lamn ("x",
        Lamn ("y",
              Lamn ("z",
                    Appn (Appn (Appn (Expn "I", Varn "x"), Varn "z"), Varn "y")))))));;
etapn_dse (lambda (x) (lambda (y) (lambda (z) (((I x) z) y)))) ->
  walk (lambda (x) (lambda (y) (lambda (z) (((I x) z) y)))) ->
  walk (lambda (y) (lambda (z) (((I x) z) y))) ->
  walk (lambda (z) (((I x) z) y)) ->
```

```

    walk (((I x) z) y) ->
    walk (((I x) z) y) <- (((I x) z) y), 0)
  y = z <-> false
etapn_dse (lambda (x) (lambda (y) (lambda (z) (((I x) z) y)))) <- None
- : (expn * int) option = None
#

```

## 5.2. Lambda terms with de Bruijn levels

Here is a stylized data type for the abstract-syntax trees of  $\lambda$  terms with de Bruijn levels, using a constant constructor `Expl` to stand for any term that is not a variable, a  $\lambda$  abstraction, or an application:

```

Inductive expl := Var1 : nat -> expl | Lam1 : expl -> expl | Appl : expl -> expl -> expl
              | Expl : expl.

```

An  $\eta$  redex of depth  $d$  reads  $\lambda\lambda\cdots\lambda e\ 0\ 1\cdots(d-1)$ , but in actuality, it contains many implicit parentheses and actually reads  $\lambda(\lambda\cdots(\lambda((\cdots((e\ 0)\ 1)\cdots)(d-1)))\cdots)$ . Reflecting these implicit parentheses, the following functions construct an  $\eta$  redex, given a term and a depth:

```

Fixpoint make_eta_redex1_aux (e : expl) (d i : nat) : expl :=
  match d with
  0   => e
  | S d' => Lam1 (make_eta_redex1_aux (Appl e (Var1 i)) d') (S i)
  end.

```

```

Definition make_eta_redex1 (e : expl) (d : nat) : expl :=
  make_eta_redex1_aux e d 0.

```

In words – `make_eta_redex1_aux` recursively constructs the resulting term with `Lam1` while accumulating the body around the given expression with `Appl`. Characteristically of  $\lambda$  terms with de Bruijn levels, a counter accounting for the current level is incremented for each instance of `Lam1`. This counter is initialized with 0.

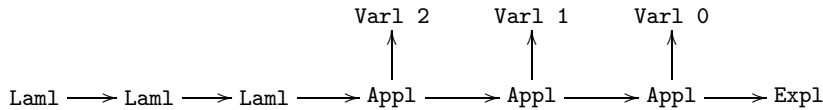
For example, evaluating `make_eta_redex1 Expl 3` yields

```

Lam1 (Lam1 (Lam1 (Appl (Appl (Appl Expl (Var1 0)) (Var1 1)) (Var1 2))))

```

Here is its abstract-syntax tree, rotated 90 degrees counterclockwise and flattened:



Detecting whether a given  $\lambda$  term with de Bruijn levels has the shape of an  $\eta$  redex fits TABA for the same reason as in Section 5.1. So likewise, we traverse the given term recursively for as long as we encounter `Lam1` constructors, starting with a level 0 and incrementing this level at each call. Then we return as soon as we encounter another constructor, and we keep returning as long as we encounter `Appl` constructors whose actual parameter is a variable whose level matches the level in effect for the matching `Lam1` constructor. (This level is visible in the lexical environment of the current recursive call.)

The following implementation is  $\lambda$  lifted and uses TABA in direct style with an option throughout, as in Section 5.1:

```
Fixpoint etapl_ds' (e : expl) (l : nat) : option (expl * nat) :=
  match e with
  | Lam1 e' => match etapl_ds' e' (S l) with
    | Some (b', d') => match b' with
      | Appl b (Var1 l') => if l' =? l
        then Some (b, S d')
        else None
      | _ => None
    end
  | None => None
  end
  | _ => Some (e, 0)
end.
```

```
Definition etapl_ds (e : expl) : option (expl * nat) :=
  match etapl_ds' e 0 with
  | Some (b, d) => if d =? 0 then None else Some (b, d)
  | None => None
  end.
```

where `etapl_ds` stands for “eta predicate for  $\lambda$  terms with de Bruijn levels in direct style.” As befit de Bruijn levels, the level is initialized at the outset and incremented for each `Lam1` constructor.

The soundness and completeness of the implementation are captured by the following theorem:

```
Theorem soundness_and_completeness_of_etapl_ds :
  forall (e : expl) (d : nat) (b : expl),
    etapl_ds e = Some (b, S d) <-> make_eta_redex1 b (S d) = e.
```

where we wrote `S d` (i.e.,  $d + 1$ ) since the depth of an  $\eta$  redex is strictly positive. In words – `etapl_ds` is sound in that it totally maps a given expression `e` to another expression and a depth, making an  $\eta$  redex with this other expression and that depth yields the given expression; and it is complete in that given an  $\eta$  redex made with a given expression and a given depth, `etapl_ds` totally yields this given expression and this given depth. This theorem is a corollary of the following lemmas, the first of which is proved by structural induction over `e` and the second by structural induction over `d`:

```
Lemma soundness_of_etapl_ds' :
  forall (e : expl) (l : nat) (b : expl) (d : nat),
    etapl_ds' e l = Some (b, d) -> make_eta_redex1_aux b d l = e.
```

```
Lemma completeness_of_etapl_ds' :
  forall (b : expl) (d l : nat) (e : expl),
    make_eta_redex1_aux b (S d) l = e -> etapl_ds' e l = Some (b, S d).
```

### 5.3. Lambda terms with de Bruijn indices

Here is a stylized data type for the abstract-syntax trees of  $\lambda$  terms, using de Bruijn indices (i.e., lexical offsets) and a constant constructor `Expi` to stand for any term that is not a variable, a  $\lambda$  abstraction, or an application:



```
Inductive expi := Vari : nat -> expi | Lami : expi -> expi | Appi : expi -> expi -> expi
  | Expi : expi.
```

An  $\eta$  redex of depth  $d$  reads  $\lambda\lambda\cdots\lambda e (d-1) (d-2)\cdots 0$ , but in actuality, it contains many implicit parentheses and actually reads  $\lambda(\lambda\cdots(\lambda((\cdots((e (d-1)) (d-2))\cdots) 0))\cdots)$ . Reflecting these implicit parentheses, the following function constructs an  $\eta$  redex, given a term and a depth:

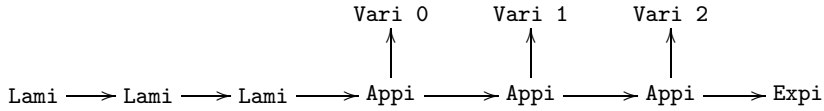
```
Fixpoint make_eta_redexi (e : expi) (d : nat) : expi :=
  match d with
  | 0 => e
  | S d' => Lami (make_eta_redexi (Appi e (Vari d')) d')
  end.
```

In words – `make_eta_redexi` recursively constructs the resulting term with `Lami` while accumulating the body around the given expression with `Appi`. Characteristically of  $\lambda$  terms with de Bruijn indices, the decreasing counter that was initialized with the desired depth can serve as the index for the successive indices of each variable in the resulting  $\eta$  redex.

For example, evaluating `make_eta_redexi Expi 3` yields

```
Lami (Lami (Lami (Appi (Appi (Appi Expi (Vari 2)) (Vari 1)) (Vari 0))))
```

Here is its abstract-syntax tree, rotated 90 degrees counterclockwise and flattened:



Detecting whether a given  $\lambda$  term with de Bruijn indices has the shape of an  $\eta$  redex fits TABA for the same reason as in Sections 5.1 and 5.2. So likewise, we traverse the given term recursively for as long as we encounter `Lam1` constructors, and then return the term that is not constructed with `Lam1` and a counter initialized with 0. Then we keep returning as long as we encounter `App1` constructors whose actual parameter is a variable whose index matches the increasing counter. Eventually, if all tests have succeeded, the result is the expression around which the  $\eta$  redex was constructed together with the depth of this  $\eta$  redex, which must be strictly positive.

The following implementation is  $\lambda$  lifted so that we can refer to the auxiliary function in lemmas. It uses TABA and a continuation to stop immediately if the given term is not an  $\eta$  redex:

```
Fixpoint etapi_cb' (e : expi) (k : expi -> nat -> option (exp1 * nat)) :=
  match e with
  | Lami e' => etapi_cb' e' (fun b' d =>
    match b' with
    | Appi b (Vari i) => if i =? d then k b (S d) else None
    | _               => None
    end)
  | _       => k e 0
  end.
```

```
Definition etapi_cb (e : expi) : option (exp1 * nat) :=
  etapi_cb' e (fun b d => if d =? 0 then None else Some (b, d)).
```

where `etapi_cb` stands for “continuation-based eta predicate for  $\lambda$  terms with de Bruijn indices.” In words – given an  $\eta$  redex of depth  $n$ , `etapi_cb'` calls itself recursively  $n$  times on the successive Lami constructors, accumulating a continuation to traverse nested applications as long as their argument is a variable whose de Bruijn index increases with the nesting of applications. The initial continuation is eventually applied to the inner expression in position of a function and to the depth of its nesting.

The soundness and completeness of the implementation are captured by the following theorem:

```
Theorem soundness_and_completeness_of_etapi_cb :
  forall (e : expi) (d : nat) (b : expi),
    etapi_cb e = Some (b, S d) <-> make_eta_redexi b (S d) = e.
```

where again we wrote `S d` since the depth of an  $\eta$  redex is strictly positive. This theorem is a corollary of the following lemmas, the first of which is proved by structural induction over `e` and the second by structural induction over `d`:

```
Lemma soundness_of_etapi_cb' :
  forall (e : expi)
    (k : expi -> nat -> option (expi * nat)),
    (exists (b : expi) (d : nat),
      etapi_cb' e k = k b d /\ make_eta_redexi b d = e)
  \ /
  (etapi_cb' e k = None).
```

```
Lemma completeness_of_etapi_cb' :
  forall (b : expi) (d : nat) (e : expi),
    make_eta_redexi b (S d) = Lami e ->
    forall k : expi -> nat -> option (expi * nat),
      etapi_cb' (Lami e) k = k b (S d).
```

An interesting aspect of this implementation is that its continuations have no free variables. Therefore defunctionalizing them yields a data type that is isomorphic to Peano numbers, and an apply function that iterates over a given Peano number. All in all, the result is a remarkably simple first-order tail-recursive predicate that foreshadows the tail-recursive variant of TABA introduced in the next section:

```
Fixpoint etapi'' (k : nat) (b : expi) (d : nat) : option (expi * nat) :=
  match k with
  | 0 => if d =? 0 then None else Some (b, d)
  | S k' => match b with
    | Appi b' (Vari i) => if i =? d then etapi'' k' b' (S d) else None
    | _ => None
  end
end.
```

```
Fixpoint etapi' (e : expi) (k : nat) : option (expi * nat) :=
  match e with
  | Lami e' => etapi' e' (S k)
  | _ => etapi'' k e 0
  end.
```

```
Definition etapi (e : expi) : option (expi * nat) :=
  etapi' e 0.
```

In words – given an expression and a counter initialized with 0, `etapi'` traverses this expression tail-recursively for as long as it encounters `Lami` constructors, incrementing the counter as it goes; then it tail-calls `etapi''` with the counter, the sub-expression that does not start with the `Lami` constructor, and an index initialized with 0; given this counter, this sub-expression, and this index, `etapi''` tail-recursively traverses the sub-expression for as long as it encounters `Appi` constructors whose actual parameter is a variable whose de Bruijn index matches the current index, until the counter reaches 0; at that point, `etapi''` checks that the resulting depth is strictly positive.

The soundness and completeness of this implementation are stated and proved *mutatis mutandis*:

```
Theorem soundness_and_completeness_of_etapi :
  forall (e : expi) (d : nat) (b : expi),
    etapi e = Some (b, S d) <-> make_eta_redexi b (S d) = e.
```

The cognoscenti will have identified that this first-order predicate implements a counter automaton. In the same vein, CPS-transforming the predicate for  $\lambda$  terms with names from Section 5.1 (see Appendix E.1), splitting its continuation into two (see Appendix E.4), and dropping the one that corresponds to the `None` case (see Appendices C and E.4) yield the implementation of a pushdown automaton with a stack of names. Small world, many disguises.

## 6. There and forth again (TAFa)

In some cases, once one gets there, there is no need to go back again: one can go forth instead to complete the computation, which makes it tail recursive throughout. This section illustrates two examples of this situation: the first is the closing exercise in the “Dear Reader” box before Section 1, and the second is due to Hemann and Friedman [10].

### 6.1. Indexing a list from the right

Indexing a list from the right means that given an index  $n$  and a list constructed as the concatenation of any list, a singleton list containing  $v$ , and any list of length  $n$ , the result should be  $v$ . If the given list is too short for the given index, the result is undefined.

#### 6.1.1. Programming

At first glance, indexing a list from the right provides another illustration of TABA: traverse the list at call time and eventually return an intermediate result initialized with the index, using the `Index` constructor below; and then at return time, decrement this index:

- if the decremented index reaches 0 before the last return, the list is long enough and the intermediate result becomes the head of the current list, using the `Found` constructor below;
- if the decremented index has not reached 0 at the last return, the list is too short for the index.

To wit:

```
type 'a intermediate_result = Index of int | Found of 'a;;

let list_index_rtl_ds vs_given n_given =
  assert (n_given >= 0);
  let rec visit vs =
    match vs with
    []      -> Index n_given
  | v :: vs' -> (match visit vs' with
                 | Index n -> if n = 0 then Found v else Index (pred n)
                 | Found v -> Found v)
  in match visit vs_given with
     | Index _ -> None
     | Found v -> Some v;;
```

To illustrate, here are two traces of the computation, one where the list is too short for the index, and then one where it is long enough:

```
# traced_list_index_rtl_ds show_int [1; 0] 5;;
visit [1; 0] ->
  visit [0] ->
    visit [] ->
      visit [] <- Index 5
    visit [0] <- Index 4
  visit [1; 0] <- Index 3
- : int option = None
# traced_list_index_rtl_ds show_int [2; 1; 0] 1;;
visit [2; 1; 0] ->
  visit [1; 0] ->
    visit [0] ->
      visit [] ->
        visit [] <- Index 1
      visit [0] <- Index 0
    visit [1; 0] <- Found 1
  visit [2; 1; 0] <- Found 1
- : int option = Some 1
#
```

In words – the given list is traversed all the way and at return time, the index is decremented. If the index reaches 0 in the course of the returns, the value to index exists and is returned.

On second thought, though, one could use the idea of Section 3.6 and decrement the given index as the given list is traversed:

- if the end of the list is reached before the index reaches 0, the list is too short for the index and the computation can stop;
- conversely, if 0 is reached first, then the list is long enough, and the length of the prefix of the given list down to the current suffix coincides with the given index; all one needs to do then is to go forth and slide through both the given list and the current suffix of the given list, preserving this length property; when the current suffix becomes its end, i.e., the empty list, the length property still holds and so the result is the head of the current list.

To wit:

```
let list_index_rtl_tafa vs_given n_given =
  assert (n_given >= 0);
  let rec there vs n =
    match vs with
    | []      -> None
    | v :: vs' -> if n = 0
                  then forth vs' vs_given
                  else there vs' (pred n)
  and forth vs trail =
    match vs with
    | []      -> Some (List.hd trail)
    | _ :: vs' -> forth vs' (List.tl trail)
  in there vs_given n_given;;
```

As one can see, given a list of length  $n$ , this implementation operates in  $n$  tail-recursive calls. (The initial tail call is not recursive.) To illustrate, here is a trace of the computation:

```
# traced_list_index_rtl_tafa show_int [1; 0] 5;;
there [1; 0] 5 ->
there [0] 4 ->
there [] 3 ->
- : int option = None
# traced_list_index_rtl_tafa show_int [3; 2; 1; 0] 1;;
there [3; 2; 1; 0] 1 ->
there [2; 1; 0] 0 ->
forth [1; 0] [3; 2; 1; 0] ->
forth [0] [2; 1; 0] ->
forth [] [1; 0] ->
- : int option = Some 1
#
```

### 6.1.2. Formalizing and proving

Let us formalize this instance of TAFE. In a nutshell,

- if the list is empty, it contains no element to index: the list is too short, no matter the index;
- if the given list is non-empty, its tail should contain at most as many elements as the given index; otherwise, the list is still too short;
- this tail can be traversed as the index is decremented until it reaches 0; once we get there, the difference between the tail of the given list and the current suffix has the same length as the given index.

We are therefore better off to first test whether the list is empty and then proceed by induction on the tail of the list if it is non-empty, a byproduct of thinking before proving, if not before programming.

Concretely:

```
Fixpoint list_index_rtl_there (V : Type) (vs_sfx : list V) (n : nat) : option (list V) :=
  match n with
  | 0 => Some vs_sfx
  | S n' => match vs_sfx with
    | nil => None
    | v' :: vs_sfx' => list_index_rtl_there V vs_sfx' n'
  end
end.
```

```
Lemma soundness_and_completeness_of_list_index_rtl_there :
  forall (V : Type) (vs' : list V) (n : nat),
    (forall vs_sfx : list V,
      list_index_rtl_there V vs' n = Some vs_sfx
    <->
      exists vs_pfx : list V,
        vs' = vs_pfx ++ vs_sfx /\ length vs_pfx = n)
  /\
  (list_index_rtl_there V vs' n = None <-> length vs' < n).
```

In words – if the given list is not empty, `list_index_rtl_there` traverses its tail, `vs_sfx`, and the given index, `n`, and returns the `n`th suffix of this tail, if one exists. (The 0th suffix of a list is itself.) The lemma is proved by structural induction over the given index.

So, after calling `list_index_rtl_there`, we have access to the first element of the given list, the tail of the given list, and the `n`th suffix of the tail of the given list, where `n` is the given index. We can then go forth and traverse the tail of the given list and the `n`th suffix until this suffix becomes the empty list. At that point, the tail has length `n` and the element just before is the element we were looking for.

Concretely:

```
Fixpoint list_index_rtl_forth (V : Type) (v : V) (vs' vs_sfx : list V) :=
  match vs_sfx with
  | nil => Some (v, vs')
  | v'' :: vs_sfx' => match vs' with
    | nil => None (* impossible case *)
    | v' :: vs'' => list_index_rtl_forth V v' vs'' vs_sfx'
  end
end.
```

The codomain is an option type because the type system does not guarantee that traversing the tail of the given list never reaches the empty list. However, this function is total:

```
Lemma soundness_and_completeness_of_list_index_rtl_forth :
  forall (V : Type) (v : V) (vs' vs_pfx vs_sfx : list V) (n : nat),
    vs' = vs_pfx ++ vs_sfx ->
    length vs_pfx = n ->
    forall op : option (V * list V),
      list_index_rtl_forth V v vs' vs_sfx = op
    <->
    exists (w : V) (ws_sfx : list V),
      op = Some (w, ws_sfx) /\
      (exists ws_pfx : list V, v :: vs' = ws_pfx ++ w :: ws_sfx) /\ length ws_sfx = n.
```

In words – given a non-empty list  $v :: vs'$  and an index  $n$  that is smaller than the length of  $vs'$ , let  $vs\_pfx$  denote the prefix of  $vs'$  of length  $n$ , and  $vs\_sfx$  denote the corresponding suffix. Since  $n$  is smaller than the length of  $vs'$ , an element  $w$  exists in  $vs'$  at index  $n$  going from right to left and so does the suffix  $ws\_sfx$  that follows it in  $vs'$ , a suffix that has length  $n$ . This element  $w$  and this suffix  $ws\_sfx$  are computed by `list_index_rtl_forth` when it is applied to  $v$ ,  $vs'$ , and  $vs\_sfx$ . The lemma is proved by structural induction over  $vs\_sfx$ .

All told, the opening description of the present section is formalized as follows:

```

Definition list_index_rtl_tafa (V : Type) (vs : list V) (n : nat) : option V :=
  match vs with
  | nil => None
  | v :: vs' => match list_index_rtl_there V vs' n with
    | Some vs_sfx => match list_index_rtl_forth V v vs' vs_sfx with
      | Some (w, ws_sfx) => Some w
      | None => None
    end
  | None => None
  end
end.

```

In words – given the empty list and an index, `list_index_rtl_tafa` returns `None`; given a non-empty list and an index, it determines whether the  $n$ th suffix of the tail of the given list exists; if it does not, `list_index_rtl_tafa` returns `None`; if it does, `list_index_rtl_tafa` computes the suffix of the given list that has length  $n$  and the element that precedes this suffix (both exist); this element is then the result of indexing the given list at the given index. The “trailing” pointer always points to a non-empty list (a type property [9]).

Soundness and completeness are a corollary of the previous lemmas:

```

Theorem soundness_and_completeness_of_list_index_rtl_tafa :
  forall (V : Type) (vs : list V) (n : nat),
    (forall w : V,
      list_index_rtl_tafa V vs n = Some w
      <->
      exists ws_pfx ws_sfx : list V,
        vs = ws_pfx ++ w :: ws_sfx /\ length ws_sfx = n)
  /\
  (list_index_rtl_tafa V vs n = None <-> length vs <= n).

```

## 6.2. Computing the common suffix of two lists

At the 2016 Scheme Workshop [10], Hemann and Friedman used TABA to compute the common suffix of two lists of unknown length in a number of calls and returns that is at most the sum of the lengths of these two lists. This computation also fits the TAFE (There and Forth Again) pattern in that the two lists can be traversed in sync (there), the longest can be slided through to reach a suffix that has the same length as the shortest (forth), and the resulting suffix and the shortest list can then be traversed in sync for comparison (again), using a total number of tail-recursive calls that is precisely the sum of the lengths of the two given lists. To ease the comparison with Hemann and Friedman’s solution, the programs in this section are expressed in Chez Scheme [29].

### 6.2.1. Finding the common suffix of two proper lists that have the same length

Given two proper lists (i.e., lists ending with nil) that have the same length, one can traverse them in parallel with an outer loop (`common-suffix_same-length`) and with an inner loop (`inner-loop`). In the outer loop, `vs` and `ws` denote the current suffix candidate, and in the inner loop, `vs_sfx` and `ws_sfx` denote their respective suffixes. (Symmetrically, we say that `vs` denotes a “trail” of `vs_sfx` and that `ws` denotes a trail of `ws_sfx`.) If this suffix is empty, then their trails both denote the resulting common suffix. Otherwise, if the heads of the respective suffixes are equal, the inner loop continues, and if they are not, a new iteration of the outer loop is initiated with two new trails, i.e., suffix candidates:

```
(define common-suffix_same-length
  (lambda (vs ws)
    (letrec ([inner-loop (lambda (vs_sfx ws_sfx)
                          (if (pair? vs_sfx)
                              (if (equal? (car vs_sfx) (car ws_sfx))
                                  (inner-loop (cdr vs_sfx) (cdr ws_sfx))
                                  (common-suffix_same-length (cdr vs_sfx) (cdr ws_sfx)))
                              vs))]
      (inner-loop vs ws))))
```

For example, here is a trace of this traversal:

```
> (traced-common-suffix_same-length '(1 2 1 2 3) '(1 2 3 2 3))
|(common-suffix_same-length (1 2 1 2 3) (1 2 3 2 3))
|(inner-loop (1 2 1 2 3) (1 2 3 2 3))
|(inner-loop (2 1 2 3) (2 3 2 3))
|(inner-loop (1 2 3) (3 2 3))
|(common-suffix_same-length (2 3) (2 3))
|(inner-loop (2 3) (2 3))
|(inner-loop (3) (3))
|(inner-loop () ())
|(2 3)
|(2 3)
>
```

In words – the outer loop is initiated with trails `(1 2 1 2 3)` and `(1 2 3 1 2 3)`. The inner loop traverses their suffix until it encounters two differing heads (namely 1 and 3). The outer loop is then re-initiated with trails `(2 3)` and `(2 3)`. The inner loop traverses their suffix to their end. The result is either of the current trails.

### 6.2.2. Traversing the suffix of a proper list to find a suffix of this list

The following procedure is given a proper list (of length  $m$ ) and one of its suffixes (of length  $n$ ), i.e., a suffix and a trail of this suffix. The procedure traverses them in parallel until the end of the suffix. It then returns the trail, which is a suffix of the given list that has length  $m - n$ :

```
(define slide
  (lambda (xs xs_sfx)
    (if (pair? xs_sfx)
        (slide (cdr xs) (cdr xs_sfx))
        xs)))
```



For example, here is a trace of this traversal:

```
> (traced-slide '(5 4 3 2 1) '(2 1))
|(slide (5 4 3 2 1) (2 1))
|(slide (4 3 2 1) (1))
|(slide (3 2 1) ())
|(3 2 1)
(3 2 1)
>
```

In words – `slide` is applied to a list of length 5 and to a suffix of this list that has length 2; it traverses them both in parallel until the end of the second list; the first list then has length  $5 - 2 = 3$  and is returned.

### 6.2.3. Finding a suffix of the longer list with the same length as the shorter list

Given two proper lists, we can enumerate their successive suffixes by traversing them in parallel:

- If both traversals end with nil, the two given lists have the same length. We can find their common suffix using `common-suffix_same-length`.
- If one traversal ends with nil but not the other, the two lists do not have the same length. We can use `slide` on the longest list and its current suffix to obtain a suffix that has the same length as the shortest list, and then `common-suffix_same-length` to find their common suffix.

Concretely:

```
(define longest-common-suffix
  (lambda (vs ws)
    (letrec ([traverse (lambda (vs_sfx ws_sfx)
                        (if (pair? vs_sfx)
                            (if (pair? ws_sfx)
                                (traverse (cdr vs_sfx) (cdr ws_sfx))
                                (common-suffix_same-length (slide vs vs_sfx) ws))
                            (if (pair? ws_sfx)
                                (common-suffix_same-length vs (slide ws ws_sfx))
                                (common-suffix_same-length vs ws)))))]
      (traverse vs ws))))
```

For example, here is a trace of this traversal:

```
> (traced-longest-common-suffix '(2 3 4 5 6 8) '(3 0 5 0 8))
|(longest-common-suffix (2 3 4 5 6 8) (3 0 5 0 8))
|(traverse (2 3 4 5 6 8) (3 0 5 0 8))
|(traverse (3 4 5 6 8) (0 5 0 8))
|(traverse (4 5 6 8) (5 0 8))
|(traverse (5 6 8) (0 8))
|(traverse (6 8) (8))
|(traverse (8) ())
|(slide (2 3 4 5 6 8) (8))
|(slide (3 4 5 6 8) ())
|(3 4 5 6 8)
```

```

|(common-suffix_same-length (3 4 5 6 8) (3 0 5 0 8))
|(inner-loop (3 4 5 6 8) (3 0 5 0 8))
|(inner-loop (4 5 6 8) (0 5 0 8))
|(common-suffix_same-length (5 6 8) (5 0 8))
|(inner-loop (5 6 8) (5 0 8))
|(inner-loop (6 8) (0 8))
|(common-suffix_same-length (8) (8))
|(inner-loop (8) (8))
|(inner-loop () ())
|(8)
(8)
>

```

In words – the two lists (here: (2 3 4 5 6 8) and (3 0 5 0 8)) are first traversed in parallel to determine that the first one is longer; what remains of the first list is a non-empty suffix of it (here: (8)); the first list and its suffix are then slid across to compute the suffix of the first list that has the same length as the second list (here: (3 4 5 6 8)); this suffix and the second list are then traversed in parallel to compute their longest common suffix. All told, and keeping in mind that initial calls are not recursive, the number of tail-recursive calls is 11, which is the sum of the lengths of the two given lists. (The author has automated this measure and verified it in practice with a variety of tests, a large number of which involved randomly generated lists.)

#### 6.2.4. All in all

An iterative solution exists for finding the common suffix of two proper lists of arbitrary length where the number of tail-recursive calls (initial calls do not count, only recursive ones) is exactly the sum of the lengths of the two given lists:

**There:** The idea is to traverse both lists in parallel until one of them is nil, which gets us there. If both are nil, then the two given lists have the same length; go to Again with both given lists. Otherwise, go to Forth with the longer list and its non-empty suffix, i.e., with the non-empty suffix and its trail.

**Forth:** The idea is to traverse both the given suffix list and its trail in parallel until the end of this suffix. The trail is then a suffix of the given longer list that has the same length as the given shorter list; go to Again with this trail and with this shorter list.

**Again:** At that point, both lists have the same length and we can traverse them in parallel with two trail pointers, resuming the Again step with new trails if the current heads of the lists are not the same. When we reach nil, each of the trails is the common suffix.

This instance of “There and Forth Again” is an optimization of “There and Back Again” that is not always applicable. When it does, though, it yields a tail-recursive solution. Here, this optimization is applicable because the order of comparisons in the list (whether from the end to the beginning or from the beginning to the end) does not matter.

## 7. Convolving lists that may not have the same length

Sometimes, it is not a mistake to convolve lists that do not have the same length, e.g., to multiply polynomials [1, Section 3], to compute Catalan numbers [1, Section 5], or to compute the Cartesian product of two sets in breadth-first order rather than the usual depth-first order [30]: one may want to convolve the shorter list with a prefix of the longer list or with a suffix of it. Since the first list determines the control flow of the convolving function, does one need to revert to the iterative solution that constructs an intermediate list? No.

### 7.1. Convolving a list and the prefix of a longer list (and vice versa)

Let  $vs$  and  $ws$  denote two lists that have the same length, and let  $xs$  denote another list.

- Should convolving  $vs$  and  $ws ++ xs$  reduce to convolving  $vs$  and  $ws$ , then the initial continuation ends up being applied to  $xs$  and a list of pairs, and it should ignore the former and return the latter.
- Should convolving  $vs ++ xs$  and  $ws$  reduce to convolving  $vs$  and  $ws$ , then the traversal of both lists should stop when reaching the end of the second list.

And indeed, convolving  $[1; 2; 3; 4]$  and  $[10; 20]$  and convolving  $[1; 2]$  and  $[10; 20; 30; 40]$  yield the same result as convolving  $[1; 2]$  and  $[10; 20]$  in this case: the suffix of the longer list is ignored, the convolving function is structurally recursive, and true to TABA the convolving function traverses the first list at call time and the second at return time.

### 7.2. Convolving a list and the suffix of a longer list (and vice versa)

Let  $vs$  and  $ws$  denote two lists that have the same length, and let  $xs$  denote another list.

- Should convolving  $vs$  and  $xs ++ ws$  reduce to convolving  $vs$  and  $ws$ , then when reaching the end of  $vs$ , one should continue to slide through the second list in the manner of Section 6, i.e., in synchrony with sliding through  $xs ++ ws$  to reach  $ws$ , and then apply the continuation to  $ws$  and the empty list.
- Should convolving  $xs ++ vs$  and  $ws$  reduce to convolving  $vs$  and  $ws$ , then the continuation should test whether its first argument is empty and keep going with the second.

And indeed, convolving  $[2; 3; 4; 5]$  and  $[40; 50]$  and convolving  $[4; 5]$  and  $[20; 30; 40; 50]$  yield the same result as convolving  $[4; 5]$  and  $[40; 50]$  in this case: the prefix of the longer list is ignored, the convolving function is structurally recursive, and true to TABA the convolving function traverses the first list at call time and the second at return time.

## 8. Conclusion

“...if you’re in any way excited by the weird and wonderful algorithms we use in functional languages to do simple things like reversing a list.” [31]

What have we learned here?

- that the TABA recursion pattern can be further illustrated, that its telling example can be refined, and that this refinement suggests alternative solutions;
- that TABA can be formalized in a way that crystallizes both its control flow and its data flow; and
- that TABA lends itself to an iterative variant, TAFE.

Proving TABA programs in direct style is carried out equationally and by structural induction. Proving TABA programs that use continuations often requires relational reasoning as well, to characterize these continuations, but equational reasoning turned out to be sufficient here.

To close, let us turn to the issue of efficiency. Is it more efficient to use TABA or to construct an intermediate data structure? The inter-derivation described in Section 4 shows that modulo any representational change of the defunctionalized continuation (e.g., going from Peano numbers to binary integers), the time complexity is the same either way. The answer therefore depends on the underlying implementation of the language processor such as unboxing polymorphic values in activation records [32]. Moving from quantitative issues to qualitative issues, TABA is noted to offer an unexpected expressive power in constrained situations [12, 8, 13]. Perhaps most significantly, however, TABA sharpens one's understanding of recursive programming, which is A Good Thing since as is often said, the sky is the limit once recursion is understood.

## Acknowledgments

Heartfelt thanks to Mayer Goldberg for the original continuation-based implementation of symbolic convolutions and for our subsequent joint study of its recursion pattern. The author is also grateful to the anonymous reviewers for perceptive evaluations and suggestions, to Bartek Klin and Damian Niminski for their editorship, and to Julia Lawall for priceless and multi-faceted comments on two versions of this article as well as for her playful formalization of Section 3 in Why3, in the course of the summer of 2020.

## References

- [1] Danvy O, Goldberg M. There and back again. *Fundamenta Informaticae*, 2005. **66**(4):397–413.
- [2] Danvy O, Millikin K. Refunctionalization at Work. *Science of Computer Programming*, 2009. **74**(8):534–549. doi:10.1016/j.scico.2007.10.007.
- [3] Fernandes JP, Saraiva J. Tools and libraries to model and manipulate circular programs. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2007). ACM Press, Nice, France, 2007 pp. 102–111. doi:10.1145/1244381.1244399.
- [4] Miranda-Perea F. Some Remarks on Type Systems for Course-of-value Recursion. *Electronic Notes in Theoretical Computer Science*, 2009. **247**:103–121. doi:10.1016/j.entcs.2009.07.051.
- [5] Sergey I. Programs and Proofs: Mechanizing Mathematics with Dependent Types. JetBrains/SPbSU Summer School, 2014. <https://ilyasergey.net/pnp-2014/>.

- [6] Shivers O, Fisher D. Multi-return function call. *Journal of Functional Programming*, 2006. **4-5**(16):547–582. doi:10.1017/S0956796806006009.
- [7] Morihatao A, Kakehi K, Hu Z, Takeichi M. Swapping Arguments and Results of Recursive Functions. In: *Mathematics of Program Construction, 8th International Conference, MPC 2006*, number 4014 in *Lecture Notes in Computer Science*. Springer, Kuressaare, Estonia, 2006 pp. 379–396. doi:10.1007/11783596\_22.
- [8] Nguyen K. Langage de combinateurs pour XML: Conception, Typage, et Implantation. PhD thesis, LRI, Université Paris Sud, Orsay, France, 2008. In English.
- [9] Foner K. ‘There and Back Again’ and What Happened After. In: *Compose Conference*, <http://www.composeconference.org/2016/program/>. 2016 [https://www.youtube.com/watch?v=u\\_0sUlwkmbQ](https://www.youtube.com/watch?v=u_0sUlwkmbQ), URL <http://www.composeconference.org/2016/program/>.
- [10] Hemann J, Friedman DP. Deriving Pure, Naturally-Recursive Operations for Processing Tail-Aligned Lists. In: *Scheme and Functional Programming Workshop, Co-located with ICFP 2016*. Nara, Japan, 2016 <http://www.schemeworkshop.org/2016/>.
- [11] Amin N, Rompf T. Collapsing Towers of Interpreters. *Proceedings of the ACM on Programming Languages*, 2018. **2**(POPL):52:1–52:33. doi:10.1145/3158140.
- [12] Brunel A, Mazza D, Pagani M. Backpropagation in the simply typed lambda-calculus with linear negation. *Proceedings of the ACM on Programming Languages*, 2019. **4**(POPL):64:1–64:27. doi:10.1145/3158140.
- [13] Wang F, Decker J, Wu X, Essertel G, Rompf T. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In: *Advances in Neural Information Processing Systems 31*, pp. 10180–10191. Curran Associates, Inc., 2018. ID:53989384.
- [14] Wang F, Zheng D, Decker JM, Wu X, Essertel GM, Rompf T. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 2019. **3**(ICFP):96:1–96:31. doi:10.1145/3341700.
- [15] Filliâtre JC. Two puzzles from Danvy and Goldberg’s “There and back again”. 2013. [http://toccata.lri.fr/gallery/there\\_and\\_back\\_again.en.html](http://toccata.lri.fr/gallery/there_and_back_again.en.html).
- [16] Bertot Y, Castéran P. *Interactive Theorem Proving and Program Development*. Springer, 2004. ISBN-10:3540208542, 13:978-3540208549.
- [17] Bird R, Wadler P. *Introduction to Functional Programming*. Prentice-Hall International, London, UK, 1st edition, 1988. ISBN-10:0134841972, 13:978-0134841977.
- [18] Burstall RM, Landin PJ. Programs and their proofs: An algebraic approach. In: Meltzer B, Michie D (eds.), *Machine Intelligence*, volume 4. Edinburgh University Press, 1969 pp. 17–43. ID:60873698.
- [19] Manna Z. *Mathematical Theory of Computation*. McGraw-Hill, 1974. ISBN-10:0070854661, 13:978-0070854666.
- [20] Nolan C. *Tenet*. Warner Bros. Pictures, 2020.
- [21] Dijkstra EW. Recursive Programming. In: Rosen S (ed.), *Programming Systems and Languages*, chapter 3C, pp. 221–227. McGraw-Hill, New York, 1960. doi:10.1007/BF01386223.
- [22] Ogori A, Sasano I. Lightweight fusion by fixed point promotion. In: *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages, SIGPLAN Notices*, Vol. 42, No. 1. ACM Press, Nice, France, 2007 pp. 143–154. doi:10.1145/1190215.1190241.

- [23] Danvy O, Schultz UP. Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. *Theoretical Computer Science*, 2000. **248**(1-2):243–287. doi:10.1016/S0304-3975(00)00054-2
- [24] Danvy O, Goldberg M. There and back again. In: Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP'02), SIGPLAN Notices, Vol. 37, No. 9. ACM Press, Pittsburgh, Pennsylvania, 2002 pp. 230–234. doi:10.1145/583852.581500.
- [25] Danvy O, Filinski A. Abstracting Control. In: Proceedings of the 1990 ACM Conference on Lisp and Functional Programming. ACM Press, Nice, France, 1990 pp. 151–160. doi:10.1145/91556.91622.
- [26] Asai K. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 2009. **22**(3):275–291. doi:10.1007/s10990-009-9049-5.
- [27] Materzok M, Biernacki D. Subtyping Delimited Continuations. In: Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages. ACM Press, Portland, Oregon, 1994 pp. 446–457. doi:10.1145/174675.178047.
- [28] de Bruijn NG. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 1972. **34**(5):381–392. doi:10.1016/1385-7258(72)90034-0.
- [29] Dybvig RK. The development of Chez Scheme. In: Proceedings of the 2006 ACM SIGPLAN International Conference on Functional Programming (ICFP'06). Invited talk. ACM Press, Portland, Oregon, 2006 pp. 1–12. doi:10.1145/1159803.1159805.
- [30] Barron DW, Strachey C. Programming. In: Fox L (ed.), *Advances in Programming and Non-Numerical Computation*, pp. 49–82. Pergamon Press, 1966. doi:10.1016/C2013-0-01911-1
- [31] Kidney DO. Typing TABA, 2020. URL <https://doisinkidney.com/posts/2020-02-15-taba.html>.
- [32] Peyton Jones SL. Personal communication at ICFP, Pittsburgh, Pennsylvania, 2002.
- [33] Danvy O. Back to Direct Style. *Science of Computer Programming*, 1994. **22**(3):183–195. doi:10.1016/0167-6423(94)00003-4.
- [34] Danvy O. Sur un Exemple de Patrick Greussay. Research Report BRICS RS-04-41, Department of Computer Science, Aarhus University, Aarhus, Denmark, 2004. doi:10.7146/brics.v11i41.21866.
- [35] Danvy O, Lawall JL. Back to Direct Style II: First-Class Continuations. In: Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, LISP Pointers, Vol. V, No. 1. ACM Press, San Francisco, California, 1992 pp. 299–310. doi:10.1145/141471.141564.
- [36] Felleisen M, Friedman DP. Control Operators, the SECD Machine, and the  $\lambda$ -Calculus. In: Wirsing M (ed.), *Formal Description of Programming Concepts III*, pp. 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986. ID:57760323.
- [37] Filinski A. Representing Monads. In: Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages. ACM Press, Portland, Oregon, 1994 pp. 446–457. doi:10.1145/174675.178047.
- [38] Johnsson T. Lambda Lifting: Transforming Programs to Recursive Equations. In: *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, Nancy, France, 1985 pp. 190–203. doi:10.1007/3-540-15975-4\_37.
- [39] Landin PJ. The Mechanical Evaluation of Expressions. *The Computer Journal*, 1964. **6**(4):308–320. doi:10.1093/comjnl/6.4.308

- [40] Peyton Jones SL. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987. ISBN-13:978-0134533339, 10:013453333X.
- [41] Reynolds JC. Definitional Interpreters for Higher-Order Programming Languages. In: *Proceedings of 25th ACM National Conference*. Boston, Massachusetts, 1972 pp. 717–740. Reprinted in *Higher-Order and Symbolic Computation* 11(4):363–397, 1998, with a foreword [42]. doi:10.1145/800194.805852.
- [42] Reynolds JC. Definitional Interpreters Revisited. *Higher-Order and Symbolic Computation*, 1998. 11(4):355–361. doi:10.1023/A:1010075320153.
- [43] Schmidt DA. State-Transition Machines for Lambda-Calculus Expressions. *Higher-Order and Symbolic Computation*, 2007. 20(3):319–332. Journal version of [45], with a foreword [44]. doi:10.1007/s10990-007-9012-2.
- [44] Schmidt DA. State-Transition Machines, Revisited. *Higher-Order and Symbolic Computation*, 2007. 20(3):319–332. doi:10.1007/s10990-007-9017-x.
- [45] Schmidt DA. State transition machines for lambda calculus expressions. In: *Semantics-Directed Compiler Generation*, number 94 in *Lecture Notes in Computer Science*. Springer-Verlag, Aarhus, Denmark, 1980 pp. 415–440. doi:10.1007/3-540-10250-7\_32.
- [46] Strachey C. *Handwritten Notes*, 1961. Archive of working papers and correspondence. Bodleian Library, Oxford, Catalogue no. MS. Eng. misc. b.267.

## A. Defunctionalization and refunctionalization

Higher-order programs are programs that use functions as values, e.g., because they involve generic functions such as `map` or `fold` or because they are in continuation-passing style. First-order programs are programs where values are first-order, i.e., are not functions. In the early 1970s [41], Reynolds proposed to ‘defunctionalize’ a particular higher-order program (an interpreter in continuation-passing style) into a first-order program (a big-step abstract machine) by representing the continuation with a data type together with an `apply` function. The key idea is that a functional value is an instance of a function abstraction. If all the function abstractions that give rise to a functional value can be enumerated, then the function space is actually a sum type, where each summand is one of these function abstractions. Each of these summands can be represented as a closure [39], i.e., a pair containing the code of the function abstraction and its lexical environment. Introducing a functional value (i.e., evaluating a function abstraction) therefore consists in constructing such a pair, and eliminating it (i.e., applying a functional value) therefore consists in extending its lexical environment with the actual parameter(s) and running its code in this extended environment. In practice, the code component is replaced by a tag that uniquely identifies it, and the `apply` function dispatches on this tag. Also, this tag is represented as a data-type constructor.

### A.1. An example of defunctionalization

Consider the continuation-passing implementation from Section 3.3:

```

Fixpoint rev2'_v3 (V : Type) (beq_V : V -> V -> bool)
  (vs : list V) (h_vs_op : list V -> bool) (ws_given : list V) : bool :=
  match vs with
  | nil => h_vs_op ws_given
  | v :: vs' => rev2'_v3 V beq_V vs' (fun ws => match ws with
    | nil => false
    | w :: ws' => if beq_V v w
      then h_vs_op ws'
      else false
    end) ws_given
  end.

```

```

Definition rev2_v3 (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) : bool :=
  rev2'_v3 V beq_V vs_given (fun ws => match ws with
  | nil => true
  | w :: ws' => false
  end) ws_given.

```

The continuation has type  $\text{list } V \rightarrow \text{bool}$ . This type is inhabited by instances of two function abstractions:

- one in `rev2_v3`: `fun ws => match ws with | nil => true | w :: ws' => ... end`, and
- one in `rev2'_v3`: `fun ws => match ws with | nil => false | w :: ws' => ... end`.

The resulting functional values are applied in the `nil` case and in the `cons` case of `rev2'_v3`.

Since continuations are instances of two function abstractions, their type can be represented using a data type with two constructors:

```

Inductive cont (V : Type) : Type := C0 : ... -> cont | C1 : ... -> cont.

```

The first function abstraction (in `rev2_v3`) has no free variables and the second one (in `rev2'_v3`) has two, `v` and `h_vs_op`. Therefore the first constructor has no argument and the second has two:

```

Inductive cont (V : Type) : Type := C0 : cont V | C1 : V -> cont V -> cont V.

```

The corresponding dispatch function and defunctionalized program read as follows:

```

Fixpoint dispatch_cont (V : Type) (beq_V : V -> V -> bool) (k : cont V) : list V -> bool :=
  match k with
  | C0 _ =>
    fun ws => match ws with nil => true | w :: ws' => false end
  | C1 _ v k =>
    fun ws => match ws with nil => false | w :: ws' => if beq_V v w
      then dispatch_cont V beq_V k ws'
      else false end
  end.

```

```

Fixpoint rev2'_v3_def (V : Type) (beq_V : V -> V -> bool) (vs : list V)
  (h_vs_op : cont V) (ws_given : list V) : bool :=
  match vs with
  | nil => dispatch_cont V beq_V h_vs_op ws_given
  | v :: vs' => rev2'_v3_def V beq_V vs' (C1 V v h_vs_op) ws_given
  end.

```



```

Definition rev2_v3_def (V : Type) (beq_V : V -> V -> bool) (vs_given ws_given : list V) :=
  rev2'_v3_def V beq_V vs_given (CO V) ws_given.

```

The alert reader will have noticed that since `dispatch_cont` returns a function, the defunctionalized program is still higher-order. However, its two invocations are fully applied, so the (higher-order) dispatch function is better defined as a (first-order) apply function:

```

Fixpoint apply_cont (V : Type) (beq_V : V -> V -> bool) (k : cont V) (ws : list V) : bool :=
  match k with
  | CO _ =>
    match ws with nil => true | w :: ws' => false end
  | C1 _ v k =>
    match ws with nil => false | w :: ws' => if beq_V v w
      then apply_cont V beq_V k ws'
      else false end
  end.

```

```

Fixpoint rev2'_v3_def' (V : Type) (beq_V : V -> V -> bool)
  (vs : list V) (h_vs_op : cont V) (ws_given : list V) : bool :=
  match vs with
  | nil => apply_cont V beq_V h_vs_op ws_given
  | v :: vs' => rev2'_v3_def' V beq_V vs' (C1 V v h_vs_op) ws_given
  end.

```

```

Definition rev2_v3_def' (V : Type) (beq_V : V -> V -> bool)
  (vs_given ws_given : list V) : bool :=
  rev2'_v3_def' V beq_V vs_given (CO V) ws_given.

```

And since the type `cont` is isomorphic to that of lists, that is how the defunctionalized continuation is represented, which leads one to the implementation of Section 3.2 where `apply_cont` is named `rev2''_v2`.

## A.2. Refunctionalization

Refunctionalization is a left inverse of defunctionalization [2]. When a data structure is constructed once and dispatched upon once, chances are it is the first-order counterpart of a higher-order function and lies in the image of defunctionalization. For example, the implementation in Section 3.1 page 125 is in defunctionalized form, since the sole reason of being for `vs_op` is to be processed in `rev2''_v1`, which de facto serves as its apply function. The refunctionalized counterpart reads as follows:

```

Fixpoint rev2'_v1_refunct (V : Type) (beq_V : V -> V -> bool)
  (vs : list V) (h_vs_op : list V -> bool) : list V -> bool :=
  match vs with
  | nil => h_vs_op
  | v :: vs' => rev2'_v1_refunct V beq_V vs' (fun ws => match ws with
    | nil => false
    | w :: ws' => if beq_V v w
      then h_vs_op ws'
      else false
    end)
  end.

```

```

Definition rev2_v1_refunct (V : Type) (beq_V : V -> V -> bool)
  (vs_given ws_given : list V) : bool :=
  rev2'_v1_refunct V beq_V vs_given (fun ws => match ws with
    nil      => true
  | w :: ws' => false
  end) ws_given.

```

### A.3. Significance of defunctionalization and refunctionalization

Reynolds's defunctionalization of a definitional interpreter that had been CPS-transformed with call by value in mind led to the CEK machine [36], and Schmidt's defunctionalization of a definitional interpreter that had been CPS-transformed with call by name in mind led to the Krivine machine [43] before either of these machines had a name. Many other abstract machines and first-order algorithms also fit as well as data structures, be they zippers or evaluation contexts. In Computer Science, do we invent things or do we discover them?

## B. Generic programming with lists

The functionals `list_fold_right` and `list_fold_left` were originally conceived by Strachey [46] to program generically over lists:

```

Definition list_fold_right (V W : Type)
  (nil_case : W) (cons_case : V -> W -> W) (vs : list V) : W :=
  let fix visit vs :=
    match vs with
    nil      => nil_case
  | v :: vs' => cons_case v (visit vs')
    end
  in visit vs.

```

```

Definition list_fold_left (V W : Type)
  (nil_case : W) (cons_case : V -> W -> W) (vs : list V) : W :=
  let fix visit vs a :=
    match vs with
    nil      => a
  | v :: vs' => visit vs' (cons_case v a)
    end
  in visit vs nil_case.

```

The first captures the ordinary recursive descent over a list (e.g., instantiating it with `nil` and `cons` yields the list-copy function) and the second the ordinary tail-recursive descent over a list using an accumulator (e.g., instantiating it with `nil` and `cons` yields the list-reverse function). Each can simulate the other by threading an accumulator.

These two functionals come handy, e.g., as a litmus test for a beginning functional programmer to demonstrate that their implementation is structurally recursive: if they cannot express it using a fold function, it isn't.

In the present case, all the structurally recursive implementations are fold-right ready, starting with the very first self-convolution functions in Section 2. Here are their fold-right counterparts:

```
let self_cnv_right vs = (* 'a list -> ('a * 'a) list *)
  let (_, ps) = list_fold_right (vs, [])
    (fun v (ws, ps) ->
      (List.tl ws, (v, List.hd ws) :: ps))
    vs
  in ps;;

let self_cnv_c_right vs = (* 'a list -> ('a * 'a) list *)
  list_fold_right (fun k -> k vs [])
    (fun v c k -> c (fun ws ps -> k (List.tl ws) ((v, List.hd ws) :: ps)))
    vs
  (fun _ ps -> ps);;
```

Likewise, the implementation in Appendix A.2 is fold-left ready. Here is its fold-left counterpart:

```
Definition rev2_v1_refunct_left (V : Type) (beq_V : V -> V -> bool)
  (vs_given ws_given : list V) : bool :=
  list_fold_left V
    (list V -> bool)
    (fun ws => match ws with
      nil => true
      | w :: ws' => false
    end)
    (fun v vs_op ws => match ws with
      nil => false
      | w :: ws' => if beq_V v w then vs_op ws' else false
    end)
    vs_given
    ws_given.
```

This counterpart illustrates the rendering of TABA using fold-left, a point recently made about convolving lists in Kidney's scientific blog [31].

## C. Lambda-lifting and lambda-dropping

Block structure and lexical scope are two cornerstones of functional programming, witness the definition of `list_fold_right` and `list_fold_left` in Appendix B, where `visit` is defined locally in the scope of `nil_case` and `cons_case`. A popular alternative is recursive equations: global mutually recursive functions with no local declarations. For example, here is an implementation of `list_fold_right` as a recursive equation:

```
Fixpoint list_fold_right' (V W : Type)
  (nil_case : W) (cons_case : V -> W -> W) (vs : list V) : W :=
  match vs with
  nil => nil_case
  | v :: vs' => cons_case v (list_fold_right' V W nil_case cons_case vs')
  end.
```

When the world was young [40], recursive equations were found to be a convenient format for compiling functional programs for the G-machine. A program transformation, lambda-lifting [38], was developed that parameterized each local function with its free variables, thus making them scope insensitive, which made it possible for them to float up to the top level and become (mutually) recursive equations. Compilers then evolved, and it was found beneficial to lambda-drop functional programs into programs with more block structure and more free variables [23], on the ground that they could be both compiled and executed more efficiently. And indeed compare the lambda-dropped version of `list_fold_right` and its lambda-lifted version, `list_fold_right'`: in the lambda-dropped version, the auxiliary function has one parameter, and in the lambda-lifted version, it has many more, most of them unchanging.

Nowadays the issue is moot for programming, since efficient compilers are known to lambda-drop source programs internally [29].

For proving, however, lambda-lifted programs are more convenient to reason about since we can only refer to entities by their name and we can only mention names that are defined globally.

## D. Tail calls, non-tail calls, lightweight fusion, and lightweight fission

Figure 3 displays four implementations to compute the first and last elements of a non-empty list. If the given list is empty, the result is `None`. Otherwise, we already know the first element, and all we need to do is to compute the last element, which is carried out tail-recursively by an auxiliary function that is defined by structural induction over the tail of the given list.

The two first implementations are lambda-dropped in that their auxiliary function is local (and `v` is declared in an outer scope), and the two last implementations are lambda-lifted in that their auxiliary function is global. The first and the third implementations are lightweight-fissioned in that the auxiliary function is invoked with a non-tail call, and the second and the fourth implementations are lightweight-fused in that the auxiliary function is invoked with a tail call. In the last implementation, `v` is passed as an extra parameter to the auxiliary function. The codomains of the auxiliary functions differ depending on whether they are fused or fissioned.

Lightweight fusion by fixed-point promotion is due to Ogori and Sasano [22]. Lightweight fission by fixed-point demotion is its left inverse.

The TAFE example in Section 6.1.2 is a good candidate for lightweight fusion:

- since `list_index_rtl_there` and `list_index_rtl_forth` are tail recursive, we can relocate the context of their initial call (i.e., the match expression in `list_index_rtl`) into their body, making the definition of `list_index_rtl` tail-recursive:

```
Definition list_index_rtl_tafa' (V : Type) (vs : list V) (n : nat) : option V :=
  match vs with
  | nil       => None
  | v :: vs' => list_index_rtl_there' V v vs' vs' n
  end.
```

```

Definition first_and_last_dropped_fissioned (V : Type) (vs : list V) : option (V * V) :=
  match vs with
  | nil      => None
  | v :: vs' => let fix aux (v' : V) (vs' : list V) : V :=
      match vs' with
      | nil      => v'
      | v'' :: vs'' => aux v'' vs''
      end
    in Some (v, aux v vs')
  end.

Definition first_and_last_dropped_fused (V : Type) (vs : list V) : option (V * V) :=
  match vs with
  | nil      => None
  | v :: vs' => let fix aux (v' : V) (vs' : list V) : option (V * V) :=
      match vs' with
      | nil      => Some (v, v')
      | v'' :: vs'' => aux v'' vs''
      end
    in aux v vs'
  end.

Fixpoint first_and_last_lifted_fissioned_aux (V : Type) (v' : V) (vs' : list V) : V :=
  match vs' with
  | nil      => v'
  | v'' :: vs'' => first_and_last_lifted_fissioned_aux V v'' vs''
  end.

Definition first_and_last_lifted_fissioned (V : Type) (vs : list V) : option (V * V) :=
  match vs with
  | nil      => None
  | v :: vs' => Some (v, first_and_last_lifted_fissioned_aux V v vs')
  end.

Fixpoint first_and_last_lifted_fused_aux (V : Type) (v v' : V) (vs' : list V) :=
  match vs' with
  | nil      => Some (v, v')
  | v'' :: vs'' => first_and_last_lifted_fused_aux V v v'' vs''
  end.

Definition first_and_last_lifted_fused (V : Type) (vs : list V) : option (V * V) :=
  match vs with
  | nil      => None
  | v :: vs' => first_and_last_lifted_fused_aux V v v vs'
  end.

```

Figure 1. Four implementations to compute the first and last elements of a non-empty list

- in `list_index_rtl'_there`, the match expression from `list_index_rtl` is relocated around the two possible end results of this tail-recursive function:

```
Fixpoint list_index_rtl'_there' (V : Type) (v : V) (vs' vs_sfx : list V) (n : nat) :=
  match n with
  | 0 => match Some vs_sfx with
        | Some vs_sfx => list_index_rtl_forth' V v vs' vs_sfx
        | None => None
      end
  | S n' => match vs_sfx with
          | nil => match (@None(list V)) with
                  | Some vs_sfx => list_index_rtl_forth' V v vs' vs_sfx
                  | None => None
                end
          | v' :: vs_sfx' => list_index_rtl'_there' V v vs' vs_sfx' n'
        end
  end.
```

- in `list_index_rtl'_forth'`, the match expression from `list_index_rtl` is relocated around the two possible end results of this tail-recursive function:

```
Fixpoint list_index_rtl'_forth' (V : Type) (v : V) (vs' vs_sfx : list V) : option V :=
  match vs_sfx with
  | nil => match Some (v, vs') with
          | Some (w, ws_sfx) => Some w
          | None => None
        end
  | v'' :: vs_sfx' => match vs' with
                    | nil => match (@None(V * list V)) with
                              | Some (w, ws_sfx) => Some w
                              | None => None
                            end
                    | v' :: vs'' => list_index_rtl'_forth' V v' vs'' vs_sfx'
                  end
  end.
```

Note how the codomain of each functions is now the same (namely that of the final result), and how `list_index_rtl'_there'` and `list_index_rtl'_forth'` take extra parameters since these parameters are now no longer available in the lexical environment, two byproducts of the program now being tail-recursive. As for the type annotations (e.g., `@None(List V)`), they were added to appease the type inferencer.

The relocated match expressions are then simplified, yielding the following tail-recursive program:

```
Fixpoint list_index_rtl'_forth'' (V : Type) (v : V) (vs' vs_sfx : list V) : option V :=
  match vs_sfx with
  | nil => Some v
  | v'' :: vs_sfx' => match vs' with
                    | nil => None
                    | v' :: vs'' => list_index_rtl'_forth'' V v' vs'' vs_sfx'
                  end
  end.
```

```

Fixpoint list_index_rtl_there'' (V : Type) (v : V) (vs' vs_sfx : list V) (n : nat) :=
  match n with
  | 0 => list_index_rtl_forth'' V v vs' vs_sfx
  | S n' => match vs_sfx with
    | nil => None
    | v' :: vs_sfx' => list_index_rtl_there'' V v vs' vs_sfx' n'
  end
end.

```

```

Definition list_index_rtl_tafa'' (V : Type) (vs : list V) (n : nat) : option V :=
  match vs with
  | nil => None
  | v :: vs' => list_index_rtl_there'' V v vs' vs' n
  end.

```

Likewise, in Section 6.2, the `slide` procedure is a candidate for lightweight fusion.

## E. A vademecum for continuations

### E.1. Direct style vs. Continuation-Passing Style (CPS)

An expression is said to be in “direct style” if its evaluation is recursive and its intermediate results are not named. For example, given appropriately typed  $f$ ,  $g$ , and  $x$ , the expression  $f (g x)$  is in direct style. An expression is in “monadic style” if its evaluation is tail recursive and its intermediate results are named. For example, assuming call by value, both  $\text{let } v1 = g x \text{ in } f v1$  and  $\text{let } v1 = g x \text{ in let } v2 = f v1 \text{ in } v2$  are in monadic style. And it is in “continuation-passing style” if its evaluation is tail recursive and if the functions it involves take an extra argument, the continuation. For example, still assuming call by value, both  $\text{fun } k \rightarrow g\_cps x (fun v1 \rightarrow f\_cps v1 k)$  and  $\text{fun } k \rightarrow g\_cps x (fun v1 \rightarrow f\_cps v1 (fun v2 \rightarrow k v2))$  are in CPS, where  $f\_cps$  and  $g\_cps$  are the continuation-passing counterpart of  $f$  and  $g$ .

For any evaluation order, any expression can be CPS-transformed using the explanation in the previous paragraph: (1) name intermediate results; (2) sequentialize their continuation by re-associating the resulting let-expressions to flatten them and  $\eta$  reduce the inner one, for the sake of proper tail recursion; and (3) introduce continuations. For example, here is the continuation-passing counterpart of `list_index_rtl_ds` in Section 6.1.1:

```

let list_index_rtl_cps vs_given n_given = (* 'a list -> int -> 'a option *)
  assert (n_given >= 0);
  let rec visit vs k = (* 'a list -> ('a intermediate_result -> 'a option) -> 'a option *)
    match vs with
    | [] -> k (Index n_given)
    | v :: vs' -> visit vs' (fun ir -> match ir with
      | Index n ->
        if n = 0 then k (Found v) else k (Index (pred n))
      | Found v ->
        k (Found v))
  in visit vs_given (fun ir -> match ir with
    | Index _ -> None
    | Found v -> Some v);;

```

The corresponding trace is a tail-recursive counterpart of that in Section 6.1.1, namely a complete series of tail calls to `visit` where continuations are accumulated, followed by a complete series of tail calls to these accumulated continuations when the list is too short, and a complete series of tail-calls to `visit` where continuations are accumulated, followed by a complete series of tail calls to these accumulated continuations when the list is long enough:

```
# traced_list_index_rtl_cps show_int [1; 0] 5;;
list_index_rtl_cps [1; 0] 5 ->
visit [1; 0] continuation_0 ->
visit [0] continuation_1 ->
visit [] continuation_2 ->
continuation_2 (Index 5) ->
continuation_1 (Index 4) ->
continuation_0 (Index 3) ->
list_index_rtl_cps [1; 0] 5 <- None
- : int option = None
# traced_list_index_rtl_cps show_int [2; 1; 0] 1;;
list_index_rtl_cps [2; 1; 0] 1 ->
visit [2; 1; 0] continuation_0 ->
visit [1; 0] continuation_1 ->
visit [0] continuation_2 ->
visit [] continuation_3 ->
continuation_3 (Index 1) ->
continuation_2 (Index 0) ->
continuation_1 (Found 1) ->
continuation_0 (Found 1) ->
list_index_rtl_cps [2; 1; 0] 1 <- Some 1
- : int option = Some 1
#
```

## E.2. Continuations and their scope

Consider the identifiers that name continuations in a continuation-passing expression. In the image of the CPS transformation, one identifier is enough: continuations are declared and then used linearly in a LIFO manner [33]. (Then defunctionalizing a continuation gives rise to a stack. And when the program in CPS is an evaluator, this stack is known as “the control stack” since Dijkstra [21].)

Sometimes, though, it is not the current continuation that is applied to continue the computation, but another one that was declared elsewhere, e.g., earlier [35]. Applying this other continuation discontinues the current computation and makes it continue elsewhere or earlier. Or sometimes, the continuation is not applied at all, indicating that it is delimited (i.e., an initial continuation is provided somewhere in the program). Not applying any continuation discontinues the current computation and makes it stop and return an intermediate result to the point where the initial continuation was provided. The control effect that is being emulated there is that of an exception in direct style (see next section for a concrete example). Many other control effects can be emulated using continuations; they give rise to control operators in direct style that achieve the corresponding control effect, e.g., delimited control [25] as well as computational monads [37].



### E.3. Continuation-passing vs. continuation-based programs

In a nutshell, the co-domain of a continuation-passing function is polymorphic and the co-domain of a continuation-based function is not.

Consider, for example, the traditional factorial function in continuation-passing style. Its continuation is linear and used in a LIFO manner:

```
let rec fac_cps n k = (* int -> (int -> 'a) -> 'a *)
  if n = 0
  then k 1
  else fac_cps (pred n) (fun a -> k (n * a));;
```

This tail-recursive implementation is interfaced with the direct-style world by supplying it with an initial continuation that is the identity function, which delimits the continuation and instantiates the type variable to `int`:

```
let fac n = (* int -> int *)
  fac_cps n (fun a -> a);;
```

Consider, for example, a tail-recursive implementation of a function that detects whether a binary tree of natural numbers with weightless nodes is a Calder mobile [34]. Its continuation is affine and used in a LIFO manner:

```
type bt = Leaf of int | Node of bt * bt;;

let rec balancedp_cb t k = (* bt -> (int -> bool) -> bool *)
  match t with
  | Leaf w -> k w
  | Node (t1, t2) -> balancedp_cb t1 (fun w1 ->
    balancedp_cb t2 (fun w2 ->
      if w1 = w2 then k (w1 + w2) else false));;

let balancedp t = (* bt -> bool *)
  balancedp_cb t (fun w -> true);;
```

This implementation is continuation-based: it is tail recursive and uses a continuation, but it only uses this continuation as long as the subtrees that are traversed so far are balanced. Otherwise, it stops, which commits the co-domain to be `bool`.

A continuation-passing program can only be tail recursive – its continuations are undelimited. A continuation-based program, on the other hand, need not be tail recursive – its continuation is delimited and therefore can be composed.

Consider, for example, a function that maps a list of unknown length  $n$  to the list of its prefixes. Its continuation is delimited and non-linear:

```
let test_prefixes candidate =
  (candidate [] = [[]]) &&
  (candidate [1] = [[]; [1]]) &&
  (candidate [2; 1] = [[]; [2]; [2; 1]]) &&
  (candidate [3; 2; 1] = [[]; [3]; [3; 2]; [3; 2; 1]]);;
```

```

let rec prefixes_cb vs k = (* 'a list -> ('a list -> 'b) -> 'b list *)
  k [] :: match vs with
    [] -> []
  | v :: vs' -> prefixes_cb vs' (fun a -> k (v :: a));;

let prefixes vs = (* 'a list -> 'a list list *)
  prefixes_cb vs (fun a -> a);;

```

Ostensibly, it proceeds in  $n$  recursive calls, even though the size of its result is quadratic in  $n$ :

```

let prefixes_gen vs = (* 'a list -> 'a list list *)
  list_fold_right (fun k -> k [] :: [])
    (fun v ih k -> k [] :: ih (fun a -> k (v :: a)))
    vs
  (fun a -> a);;

```

The key is to compose each of its successive continuations to construct the successive prefixes, something that can also be achieved in CPS by layering continuations [25]:

```

let rec prefixes_cps vs k mk =
  (* 'a list -> ('a list -> ('b -> 'c) -> 'c) -> ('b list -> 'c) -> 'c *)
  k [] (fun p ->
    match vs with
      [] -> mk (p :: [])
    | v :: vs' -> prefixes_cps vs'
      (fun p mk -> k (v :: p) mk)
      (fun ps -> mk (p :: ps)));;

let prefixes' vs = (* 'a list -> 'a list list *)
  prefixes_cps vs (fun p mk -> mk p) (fun ps -> ps);;

```

## E.4. Splitting continuations

Based on the type isomorphism between  $A + B \rightarrow C$  and  $(A \rightarrow C) \times (B \rightarrow C)$ , we can split the continuation into two in the definition of `list_index_rtl_cps` from Appendix E.1:

```

let list_index_rtl_cps2 vs_given n_given = (* 'a list -> int -> 'a option *)
  assert (n_given >= 0);
  let rec visit vs k_Index k_Found = (* 'a list -> (int -> 'a option) ->
    ('a -> 'a option) -> 'a option *)
    match vs with
      [] -> k_Index n_given
    | v :: vs' -> visit vs'
      (fun n -> if n = 0 then k_Found v else k_Index (pred n))
      (fun v -> k_Found v)
  in visit vs_given
    (fun _ -> None)
    (fun v -> Some v);;

```

where we can also  $\eta$  reduce `fun v -> k_Found_it v` into `k_Found_it`. Instead of passively threading `k_Found_it` from its point of definition to its point of use, we can drop this parameter from its point of definition to its point of use and simplify its application [23]:

```

let list_index_rtl_cb vs_given n_given = (* 'a list -> int -> 'a option *)
  assert (n_given >= 0);
  let rec visit vs k = (* 'a list -> (int -> 'a option) -> 'a option *)
    match vs with
    | [] -> k n_given
    | v :: vs' -> visit vs' (fun n -> if n = 0 then Some v else k (pred n))
  in visit vs_given (fun _ -> None);;

```

In this parameter-dropped definition, `visit` only uses its continuation until `n` denotes 0, witness the following trace that features

- a complete series of tail calls to `visit` where continuations are accumulated, followed by a complete series of tail calls to these accumulated continuations when the list is too short, and
- a complete series of tail-calls to `visit` where continuations are accumulated, followed by an interrupted series of tail calls to these accumulated continuations when the list is long enough:

```

# traced_list_index_rtl_cb show_int [2; 1; 0] 5;;
list_index_rtl_cb [2; 1; 0] 5 ->
visit [2; 1; 0] continuation_0 ->
visit [1; 0] continuation_1 ->
visit [0] continuation_2 ->
visit [] continuation_3 ->
continuation_3 5 ->
continuation_2 4 ->
continuation_1 3 ->
continuation_0 2 ->
list_index_rtl_cb [2; 1; 0] 5 <- None
- : int option = None
# traced_list_index_rtl_cb show_int [5; 4; 3; 2; 1; 0] 3;;
list_index_rtl_cb [5; 4; 3; 2; 1; 0] 3 ->
visit [5; 4; 3; 2; 1; 0] continuation_0 ->
visit [4; 3; 2; 1; 0] continuation_1 ->
visit [3; 2; 1; 0] continuation_2 ->
visit [2; 1; 0] continuation_3 ->
visit [1; 0] continuation_4 ->
visit [0] continuation_5 ->
visit [] continuation_6 ->
continuation_6 3 ->
continuation_5 2 ->
continuation_4 1 ->
continuation_3 0 ->
list_index_rtl_cb [5; 4; 3; 2; 1; 0] 3 <- Some 3
- : int option = Some 3
#

```

Discontinuing the computation is characteristic of encoding an exception in direct style. Exceptions in OCaml, however, are global and monomorphic, so to preserve the polymorphism of the corresponding direct-style implementation, one needs to resort to a parameterless exception and a local reference:

```

exception Found_it;;

let list_index_rtl_dse vs_given n_given =
  assert (n_given >= 0);
  let optional_result = ref None
  in try let rec visit vs =
      match vs with
      | []      -> n_given
      | v :: vs' -> let n = visit vs'
                    in if n = 0
                       then (optional_result := Some v;
                             raise Found_it)
                       else pred n
    in let _ = visit vs_given
    in None
  with Found_it -> !optional_result;;

```

The traces are as expected, namely

- a complete series of recursive calls to `visit` followed by a complete series of returns when the list is too short, and
- a complete series of recursive calls to `visit` followed by an interrupted series of returns when the list is long enough:

```

# traced_list_index_rtl_dse show_int [3; 2; 1; 0] 10;;
visit [3; 2; 1; 0] ->
  visit [2; 1; 0] ->
    visit [1; 0] ->
      visit [0] ->
        visit [] ->
          visit [] <- 10
        visit [0] <- 9
      visit [1; 0] <- 8
    visit [2; 1; 0] <- 7
  visit [3; 2; 1; 0] <- 6
- : int option = None
# traced_list_index_rtl_dse show_int [5; 4; 3; 2; 1; 0] 3;;
visit [5; 4; 3; 2; 1; 0] ->
  visit [4; 3; 2; 1; 0] ->
    visit [3; 2; 1; 0] ->
      visit [2; 1; 0] ->
        visit [1; 0] ->
          visit [0] ->
            visit [] ->
              visit [] <- 3
            visit [0] <- 2
          visit [1; 0] <- 1
        visit [2; 1; 0] <- 0
      - : int option = Some 3
    #

```

Dear Reader:

Thanks for getting acquainted further with “There and Back Again.” As a farewell gift, here are some more programming exercises.

**Reversing a list, again:**

Given a list of length  $n$ , where  $n$  is unknown, construct its reverse in  $n$  recursive calls without using an accumulator. Not using an accumulator means that when expressing your solution using either fold functional for lists (i.e., `list_fold_right` or `list_fold_left`, they give the same result here), the first argument of the fold functional should not be a function.

**Deciding whether a list is a self-convolution:**

Given a list of pairs of length  $n$ , where  $n$  is unknown, determine whether this list represents a self-convolution in  $n$  recursive calls.

**Decomposing a symbolic convolution into its two components:**

Given a symbolic convolution  $[(x_1, y_n), (x_2, y_{n-1}), \dots, (x_{n-1}, y_2), (x_n, y_1)]$ , where  $n$  is unknown, construct its two components  $[x_1, x_2, \dots, x_n]$  and  $[y_1, y_2, \dots, y_n]$  in  $n$  recursive calls.

**Swapping parity-indexed elements in lists of odd length:**

Implement a function that, given a list of odd length, swaps its even-indexed elements, so that, e.g., `[0; 1; 2; 3; 4; 5; 6]` is mapped to `Some [6; 1; 4; 3; 2; 5; 0]`, and another that, given a list of odd length, swaps its odd-indexed elements, so that, e.g., `[0; 1; 2; 3; 4; 5; 6]` is mapped to `Some [0; 5; 2; 3; 4; 1; 6]`. Given a list of length  $n$ , where  $n$  is unknown, the two functions should proceed in  $n$  recursive calls. Any list of even length should be mapped to `None`.

**Swapping parity-indexed elements in lists of even length:**

Guess what.

**Twice as fast:**

Assume a function `rev_stutter2` that maps a list  $[x_1, x_2, \dots, x_{n-1}, x_n]$  to  $[x_n, x_n, x_{n-1}, x_{n-1}, \dots, x_2, x_2, x_1, x_1]$ . Given two lists, the first one of length  $n$ , where  $n$  is unknown, detect whether the second list is the result of applying `rev_stutter2` to the first list, in  $n$  recursive calls.

At least two solutions exist: one that recurses on the first list, and another that recurses on the second list.

Do feel free to share your solutions with the author by email, just for the joy of functional programming and proving. There are no open problems here: the point of these exercises is that if you can solve any of them, you do get TABA.