Rewinding functions through CPS*

An experience report

Marco Giunti
NOVA LINCS, Universidade NOVA de Lisboa
Portugal

ABSTRACT

The ability to rewind or step back computations is fundamental in many functional programming approaches, which include the teaching of foundational computing courses, by allowing step-wise execution of algorithms, and the debugging of functional applications, by permitting the inspection of the intermediate values of recursive computations. In this report, we present a functional programming pearl that shows how this can be accomplished in continuation-passing style (CPS) by tracing the argument and the continuation of each recursive call. To obtain maximum generality, the mechanism is provided by means of a monad transformer that adds trace functionalities to arbitrary monads. We show applications of the transformation involving algebraic types, data structures, references, and exceptions, and conclude by presenting a functor that provides support for the automatic generation of CPS functions with rewind functionalities.

ACM Reference Format:
Marco Giunti. 2019. Rewinding functions through CPS: An experience report. In *Proceedings of the 21th International Symposium on Principles and Practice of Declarative Programming, Porto, Portugal, October 07 - 09, 2019.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, the functional programming paradigm has became an hot topic among developers, largely because of the the industry's support for functional features in mainstream programming languages, being Oracle's introduction of lambda expressions in Java 8 the most notable example. Still, the most part of developers seem to be reluctant to adopt the functional style of programming in their projects: this is matter of a long-standing debate in the programming language community and in academia, where functional programming historically has a strong support. Without entering into the debate, we identify at least two aspects that we believe relevant to the diffusion of the functional programming paradigm among developers: best practices of teaching formal models and functional programming in high level education, and support for the functional software development in the day-to-day practice.

This experience report aims at presenting a contribution towards these directions by introducing a programming pearl that shows how to offer support for tracing functionalities in continuation-passing style (CPS) programming (c.f. [1–3, 10]). We believe that this is relevant for the aforementioned objectives: from the point of view of education, support for tracing in CPS may allow students to better understand the behaviour of algorithms by stepping back and forward the execution: this will be experimented in the context of the project *FACTOR* (http://ctp.di.fct.unl.pt/FACTOR), where tools based on the construction presented in this paper will support the teaching of formal languages as automata and logics; from the point of view of the daily routine of a developer, these added functionalities may be of help in the debugging of programs by providing a general solution to the tracking of CPS calls.

To illustrate, let us consider the problem of debugging a CPS variant of the Fibonacci function in OCaml, that is `fib : int -> (int -> int) -> int`. The two main (native) debugging tools available are ocamltop's `trace`, and `ocamldebug`; however, both of them offer poor support for CPS. For instance, the tracing of `fib 6 (fun x -> x)` in ocamltop produces 100 lines: the first 75 lines contain a sequence of entries of the form `<-- n, fib --> <fun>, fib* <-- <fun>` where $0 \leq n \leq 6$; the remaining 25 lines replicate the entry `fib* --> 13`. From this information, we can only infer the sequence of arguments n evaluated in the call's tree, and the final value: that is, the information embedded in the CPS continuation is lost. To better analyse the behaviour of CPS evaluation, we may be interested in applying the continuations to their argument in each step of the evaluation, thus providing support for some form of reverse engineering based on the values produced in the intermediate evaluation steps.

In this report, we present a general solution to this problem by providing a monad transformer that adds trace and rewind functionalities to arbitrary monads. The *Log-Rewind* monad transformer provides a general framework to analyse the step-by-step execution of CPS transformations by allowing to record the pair of arguments `(n, f)` passed in each call, where `n` has type `'a`, and `f` is a function from `'a` to a monad with parametric type `('a, 'b)`: that is, `n : 'a` and `f : 'a -> ('a, 'b) mon`. For instance, the execution of a variant of `fib` that uses a transformation of the State monad by passing the argument 6, and an " identity" monadic function as continuation, generates a log that allows to produce a sequence of 25 pairs of the form `(n, f n)`; an excerpt of this sequence is (pretty-print): (6, 6), (5, 10), (4, 12), (3, 13), ... , (0, 12), ... , (2, 13), (1, 13), ... , (0, 12).

We further increase the support to the software development process by introducing a functor that allows to automatically generate the code for Log-Rewind CPS functions. The monadic variant

*This work was partially supported by the Tezos foundation through a grant for the project "FACTOR - A Functional Programming Approach to Teaching Portuguese Foundational Computing Courses".

PPDP 2019, 7-9 October 2019, Porto, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of `fib` is generated by abstracting the structure of the recursion by means of algebraic constructors for binary and unary functions, and self-calls, as illustrated by the list below [1]:

```
[(Binary (+), (fun n -> n - 1), Self) ;
 (Unary (fun n -> n), (fun n -> n - 2), Self)]
```

Contribution. We present a functional programming pearl that shows how to add trace and rewind functionalities to CPS functions. The mechanism is deployed by means of a monad transformer that adds trace, forward, and backward values to arbitrary monads. We analyse some application of our construction, and conclude by presenting a functor that allows to automatically generate CPS functions with support for log and rewind from the abstract structure of the idealized function.

Related Work. The use of a replay monad to trace intermediate computations is not new, e.g. [4, 9, 11]; however, to the best of our knowledge, this is the first attempt to devise a general mechanism to add tracing functionalities to CPS functions. Log-based replay has been used in Haskell to implement server-side web scripting through CGI [11]. In that framework, a CGI monad (implemented on top of the IO monad) mediates the interaction with the environment by means of functions `tell` and `ask`, and relies on a log containing all interactions of the server with the client page to invoke the callback actions included in a `ask` invocation. A monad transformer is used in [9] to accumulate observations about execution as an effect. An observation is a monadic value `mark`, that can be interpreted as a tick; this increase the flexibility of the approach by deferring the instantiation of the observation to the transformation. The paper [4] shows how to implement delimited continuations in terms of exceptions and state. To this aim, they consider a notion of replayed computation based on shift and reset, and on list of frames (i.e. the past and the future). The run and the bind rely on a stack of choice indexes of the form (index, length); the indexes provide the low-level representation of the past and future in terms of a computation tree, and allow to execute a replay-based nondeterminism simulation algorithm.

2 LOG-REWIND TRANSFORMER

The Log-Rewind monad transformer is presented in Table 1; we use the OCaml programming language, while the construction can be ported to functional languages with support for functors. We consider a type signature for monads `M` that contain the values `ret` (*return*), `bind`, and `run` [6, 8, 12], thus making explicit that a monad of interest must be able to produce an observable result.

The monad transformer [7] is a functor that generate monads with the ability to log entries of type `('a, 'b) argFunPair = ('a * ('a -> ('a, 'b) mon)` (c.f. lines 12–14) ; each pair contains an argument of type `'a` and a continuation of type `('a -> ('a, 'b) mon)`: that is, the type of the log is `('a, 'b) log = ('a, 'b) argFunPair list`. This involves a mutually recursive definition, as type `('a, 'b) mon` is defined as a function that receives a value of type `('a, 'b) log` and returns a value of type `('a * ('a, 'b) log) M.mon`, where `M` is the monad to be transformed. Functions `ret` and `bind` (lines 16–17) transform the respective monad's functions

[1] The structure roughly abstracts the code of a pure function of the form: `let f n = if ... else f (n - 1) + f (n -2)`. The full structure is more involved and includes information about the base of the recursion: see § 4 for all the details.

Table 1: Log-Rewind monad transformer

```
1  module type MONAD = sig
2      type 'a mon
3      type 'a result = 'a
4      val ret: 'a -> 'a mon
5      val bind: 'a mon -> ('a -> 'b mon) -> 'b mon
6      val run: 'a mon -> 'a result
7    end
8  (* Monad transformer : adding Log-Rewind to monad M *)
9  (* Log contains pairs of (argument, function) where *)
10 (* function receives argument and returns a monad  *)
11 module LogRwdTransf(M : MONAD) = struct
12     type ('a, 'b) log = ('a, 'b) argFunPair list
13      and ('a, 'b) mon = ('a, 'b) log -> ('a * ('a, 'b) log) M.mon
14      and ('a, 'b) argFunPair = ('a * ('a -> ('a, 'b) mon))
15     type ('a, 'b) result = ('a * ('a, 'b) log) M.result
16     let ret x = fun l -> M.ret (x, l)
17     let bind m f = fun l -> M.bind (m l) (fun (x, s') -> f x s')
18     let (>>=) = bind
19     let lift m = fun l -> M.bind m (fun x -> M.ret (x, l))
20     let run (m: ('a, 'b) mon) : ('a, 'b) result =
21       M.run (M.bind (m []) (fun (x, l) -> M.ret (x, List.rev l)))
22     let log (p : ('a, 'b) argFunPair) = fun l -> M.ret ((), p :: l)
23     let sizeOfLog (s : ('a, 'b) result) = match s with _, l -> List.length l
24     let nthFunPair (s : ('a, 'b) result) (n : int) = match s with _, l ->
25         if n < 0 || n >= sizeOfLog s then List.nth l 0 else List.nth l n
26     let navigate (s : ('a, 'b) result) (n : int) =
27       match nthFunPair s n with x, f ->
28         match run (f x) with r, _ -> s, r, n
29     let backward (s : ('a, 'b) result) (n : int) =
30       let  k = sizeOfLog s
31       in navigate s (if n < 0 || n > k then k - 1 else n - 1)
32     let forward (s : ('a, 'b) result) (n : int) =
33       let  k = sizeOfLog s
34       in navigate s (if n < 0 || n > k then 0 else n + 1)
35   end
```

by adding the log `l` to the result. Function `lift` (line 19) is used to transform monad's values `m` that are different from `ret`, `bind` and `run`. The `run` (line 20) provides an initial empty log and returns a result of the form `M.ret (x, l)`, where `x` is the value produced by the monad and `l` is the log, conveniently reversed.

The tracing of intermediate computations and the ability to step back and forward among the results are provided by functions `log` (line 22), `backward` (line 29), and `forward` (line 32), respectively. Function `log` allows to record and entry of the form `(x, f)`, where `x` is the argument and `f` is the (monadic) continuation. Function `backward` receives a result (produced by the `run`) and a position, and steps back to the previous position in the log by means of function `navigate`; a sanity check on the index is provided in order to avoid index out of bounds. Function `navigate` (line 26) is the heart of the step back and forward mechanism: it receives a result and a position `n`, extract the pair `(x, f)` from the log contained in the result, runs the monad `f x`, and returns the result of the run. Function `forward` allows to step forward the computation by using the same mechanism of `backward`, with appropriate indices.

In the next section, we analyse applications of the Log-Rewind monad transformer to the State monad. We outline here the code of the monad transformation, where we are considering a store-based implementation of the State monad using references [6].

```
module StateLogRwd = struct
    include LogRwdTransf(State)
    let ref x = lift (State.ref x)
    let deref r = lift (State.deref r)
    let assign r x = lift (State.assign r x)
  end
```

Table 2: Log-Rewind CPS evaluation of expressions

```
1  let log_evalCps x f =
2    let f_neutral = function  And _ -> Top | Or _ -> Bot | _ -> Nan
3    in StateLogRwd.(run
4       (let rec eval e k =
5          (match e with Prop p -> log (p, k) | _ -> log (f_neutral e, k))
6          >>= fun _ ->
7          match e with
8          | Prop p -> k p
9          | And (e1, e2) -> eval e1 (fun z1 -> eval e2 (fun z2 -> k (andT z1 z2)))
10         | Or (e1, e2)  -> eval e1 (fun z1 -> eval e2 (fun z2 -> k (orT z1 z2)))
11         | Neg e -> eval e (fun z -> k (negT z))
12         | _ -> raise OpenFormula
13       in (eval x f)))
```

3 APPLICATIONS

In this section, we show applications of Log-Rewind monad transformer of Table 1 involving algebraic types, data structures, references, and exceptions. To this aim, we consider the `StateLogRwd` monad (see above) obtained by transforming the State monad with the Log-Rewind functor, and subsequently add exceptions.

We start with a classic example, that is the Fibonacci function.

```
let log_fibCps x f =
  StateLogRwd.
  (run (let rec fib n k =
         (if n >= 0 then log (n, k) else log (1, k)) >>= fun _ ->
          if n <= 1
          then  k 1
          else  fib (n - 1) (fun n1 -> fib (n-2) (fun n2 -> k (n1 + n2)))
        in fib x f))
```

The definition of the CPS variant of Fibonacci, named `fib` above, is unsurprising: the only difference w.r.t. the "standard" CPS version is that we use the `log` function in order to trace pairs of the form `(n, k)`, where `n` is the argument and `k` is the (monadic) continuation, and the `bind` function to "connect" the log operation with the remainder of the computation. This can be done because `fib` is encapsulated in `StateLogRwd.run`, that is the body of the run is the application `fib x f`, where `x` and `f` are provided by the top-level function, `log_fibCps`. We note that function `fib` accepts the same parameters of `log_fibCps` and produces a monad of type `(int, 'b) StateLogRwd.mon`, that is the argument of function `run`. To instantiate `log_fibCps`, we pass an integer and the identity function `fun x l -> StateLogRwd.ret x l`; the call produces an entry of type `(int, 'b) StateLogRwd.result` of the form `(y, g)`, where `y` is an integer (i.e. the Fibonacci number), and `g` is the log.

Given a result, the monad `StateLogRwd` offers (at least) two ways to see intermediate computations (c.f. Table 1): explicit, by extracting the log from the result and by accessing to its entries of type `('a, 'b) argFunPair`; implicit, by using functions `backward` and `forward`. We provide below an example of going backward, where `rm` is a reference to `(y, g)`, `f` is a formatting function, and `p` is the pretty-printer; the code to move forward is obtained by passing the initial position `-1` as second argument of `forward`.

```
let rn  = ref StateLogRwd.(sizeOfLog !rm) in
for i = 0 to !rn - 1 do
  let m, r, n = StateLogRwd.(backward !rm !rn) in
  Format.printf "Position :%d Value :%a\n" (!rn - 1) (p f) r; rn := n; rm := m
done
```

Algebraic types. The next experiment consists in study the impact of variant types on our construction. To this aim, we consider a non-trivial application, that is the evaluation of propositional formulae. The case study is interesting because CPS evaluation is more involved, and because the choice of the argument to log is not straightforward, as we will show. Last but not least, this case study has a special interest from the point of view of education, as it provides a first test of the use of the Log-Rewind mechanism to develop tools to support the teaching of foundational computing courses, which is the main subject of the FACTOR project (cf. § 1).

We consider an algebraic representation `t` of type *Bool* where the constructor $\top$ corresponds to true and the constructor $\bot$ corresponds to false. The algebra introduces a third construct, **N**, read as "None", which intuitively acts as neutral of all operations defined over `t`; this will be used in logging partial CPS evaluations, as described below. The algebraic commutative operations $\wedge$, $\vee$, and the operation $\neg$ over `t` are defined as expected [2], modulo **N**:

$$\text{AndTop}\ \frac{y = \top \text{ or } y = \mathbf{N}}{\top \wedge y = \top} \qquad \text{AndBot}\ \frac{}{\bot \wedge y = \bot}$$

$$\text{OrTop}\ \frac{}{\top \vee y = \top} \qquad \text{OrBot}\ \frac{y = \bot \text{ or } y = \mathbf{N}}{\bot \vee y = \top}$$

$$\text{negTop}\ \frac{}{\neg\bot = \top} \qquad \text{negBot}\ \frac{}{\neg\top = \bot} \qquad \text{negN}\ \frac{}{\neg\mathbf{N} = \mathbf{N}}$$

Propositional formulae (hereafter referred as expressions) are obtained by adding variables to the algebra, resulting in the type `boolE` built upon the constructor `Prop` carrying a value of type `t`, `Var` carrying a value of some type `i`, and the recursive constructors `And`, `Or`, and `Neg`, which are the direct counterpart of the operators $\wedge$, $\vee$ and $\neg$ over `t`. Evaluation of a (closed) expression produces a value of type `t`, where closed means that the expressions is `Var`-free.

Function `log_evalCps` in Table 2 implements the Log-Rewind evaluation of expressions by logging the pairs `(p, k)` passed to each call of the inner recursive function `eval`, where `p` has type `t` and `k` has type `t -> (t, 'b) mon`. Lines 4–12 show how such function is built; we note that the continuations corresponding to the binary and the unary inductive cases have a structure similar to those of Fibonacci and of the factorial, respectively. However, as anticipated, there is a fundamental difference w.r.t. the factorial and Fibonacci, that is that those functions are integer operations, while expression evaluation is not an operation. This is relevant since `eval` receives an argument of type `boolE`, while a log entry has type `(t, t -> (t, 'b) mon)`: that is, we need to produce a value of type `t` given an expression of type `boolE`.

Naively, we could record the proper value when `e` is a proposition, and the default value **N** otherwise; this accounts for inhibiting any analysis on the corresponding log entries, and is less satisfactory. The solution adopted in line 5 of Table 2 allows some improvement by recording the (proper) neutral element w.r.t. the constructor of the expression; in this way, we can at least infer the nature of the expression received in input. This design choice motivates the introduction of the constructor **N** as neutral for the negation, that is if we find an entry `(N, k)` in the log, then we know that the corresponding expression received by `eval` was of the form `Neg e`. Interestingly, this solution also simplifies the implementation of an automatic generator of Log-Rewind CPS evaluators, as we will discuss in the next section.

Data structures, references, and exceptions. Consider a function `delete` to remove an element from a binary search tree, mutually recursively defined with a function `join` to glue the two sub-trees

[2] In the implementation, the operations are called `andT`, `orT`, and `negT`, respectively.

Table 3: Generator of Log-Rewind CPS-evaluators (excerpt)

```
1  module GenerateLogRwdCps(M :  MONAD) = struct
2      module LogRwdMonad = struct include LogRwdTransf(M) end
3      type ('a, 'b) func = Unary of 'a -> 'b | Binary of a -> 'a -> 'b | Self
4      let genCps f_base pred_base tree_shold f_ind f_neutral f_args =
5        fun x f -> let default = f_neutral x in
6        LogRwdMonad.(run
7           (let rec k n p  =
8              if pred_base n tree_shold
9              then log (f_base n, p) >>= fun _ -> p (f_base n)
10             else
11               log (f_ind n, p) >>= fun _ ->
12               let rec gen_f acc = function ... in
13               let gen_ = function
14                 | (opf, x, gf) :: t ->
15                    (match gf with
16                     | Self -> ...
17                     | Unary g  ->
18                        (match t with
19                         | [] -> ...
20                         | _ ->
21                            (match opf with
22                             | Binary op ->
23                                (gen_f [(op, default)] t) (op (g (x n)) default))))
24                 | _ -> raise (EmptyList __LOC__)
25               in gen_ (f_args !actual_x)
26           in k x f))
27   end
```

once the element is found, and raising `Not_found` otherwise while indicating the closest element found. In order to devise a Log-Rewind transformation, we need to use currying to isolate the CPS function with signature `'a bst -> ('b -> 'a bst) -> 'a bst`, and to define monadic variants of `delete` and `join` that both log entries of the form `('a bst, 'a bst -> ('a, 'b) mon)`. Functions `ref`, `deref`, and `assign` are already available in the State monad (c.f. § 2). Functions `raise` and `trywith` are added to monadic computations by means of an Exception transformation of the state monad, and then by a Log-Rewind transformation of the obtained monad (c.f. § A).

```
let log_delete el t f s =
LogStateException.(run (ref el >>= fun rf ->
  let rec delete = fun x ->
  let rec delete_ t k = log (t, k) >>= fun _ ->
    match t with
    | Empty -> deref rf >>= fun el ->
               raise (Not_found (Format.sprintf "Closest element :%a\n" s el))
    | Node (l, y, r) -> assign rf y >>= fun _ ->
       if x = y then join l r k else
         if x < y then delete_ l (fun lw -> k (Node (lw, y, r))) else ...
  and join l r k = match l, r with
    | Empty, r -> log (r, k) >>= fun _ -> k r | ...
    | l, r -> let m = find_max l in (delete m) l (fun lw -> k (Node (lw, m, r)))
  in delete_
  in delete el t f))
```

4 AUTOMATED GENERATION OF LOG-REWIND CPS-EVALUATORS

The examples studied in § 3 show that CPS evaluation with Log-Rewind support has a common structure, that is there is a pattern for the evaluation. In this section, we present a functor that allows to automatically generate Log-Rewind CPS evaluators for a class of functions that can be described algebraically by means of unary and binary operators. As in the previous section, we start by considering functions using base types, and subsequently study the impact of pattern matching.

To illustrate, consider a non-tail recursive function to calculate the factorial of an integer:

```
let fact x = if x <= 1 then 1 else x * fact (x - 1)
```

We abstract the recursive structure of the function by using constructors for binary and unary operators:

Table 4: Automatic generation of Log-Rewind factorial

```
1  (* Automatically generated Log-Rewind CPS *)
2  let gen_factCps =
3    GenCpsState.(genCps
4        (fun _ -> 1) (<=) 1 (fun x -> x) (fun _ -> 1)
5        (fun _ -> [(Binary ( * ), (fun n -> n), (Unary (fun x -> x))) ;
6                   (Unary (fun x -> x), (fun n -> n - 1), Self)]))
7  (* Programmed Log-Rewind CPS  *)
8  let factCps x f =
9    StateLogRwd.
10   (run (let rec fact n e =
11           (if n > 0 then log (n, e) else log (1, e)) >>= fun _ ->
12           if n < 2
13           then  e 1
14           else  let g = fun n1 -> e (n1 * n)
15                 in fact (n-1) g
16         in (fact x f)))
```

```
[(Binary ( * ), (fun n -> n), (Unary (fun x -> x))) ;
 (Unary (fun x -> x), (fun n -> n - 1), Self)]
```

The `Binary` constructor in the first position of the first entry indicates the binary operator to apply to the terms abstracted by the first and by the second entry. Roughly, this corresponds to the code `n * f m` where `f` and `m` are taken from the next entry. The `Unary` constructor in the first position of the second entry is applied to the term abstracted by the entry itself (since it is unary), which roughly corresponds to `fact (n - 1)`, that is `f = fact` and `m = n - 1`. The `Self` constructor in the third position of the second entry indicates a recursive self-call with argument provided by the function in the second position of the entry, while the `Unary` constructor in the third position of the first entry indicates to apply the unary operator to the argument provided by the function in the second position of the entry.

Table 3 provides more details on this mechanism by presenting an excerpt of the functor that automatically generates Log-Rewind CPS evaluators (c.f. § A). Function `genCps` receives six arguments, which are (in order) : a *function* that returns the *base value*, a *predicate* that identifies the *base of the recursion*, a *treshold* to be used by the predicate, a *function* that provides the *value to be logged,* a *function* that provides the *neutral element* w.r.t. the first operation in the last argument, a *function* that returns a *list* that represent the *abstract structure of the recursion*. Incidentally, we note that defining such arguments as functions provide support for (monadic) pattern matching; an example is briefly discussed in the next paragraph.

We illustrate the rationale behind function `genCps` by means of the factorial example. Table 4 shows a call to `GenCpsState.genCps` that generates a Log-Rewind CPS for the factorial, where the module includes the signature `GenerateLogRwdCps(State)`. The call should produce a function equal (by extensionality) to `factCps` in the same Table [3]. Consider the call in line 3 of Table 4 and contrast the lines of code in Table 3. The value returned by the call is a function which body is partially generated by the inner function `gen_` , which acts as a wrapper for the recursive function `gen_f`, which we totally omit. The call in line 25 of Table 3 passes as argument to `gen_` the list with the structure, which pattern matches in lines 17, 22 and returns the value

```
(gen_f [(( * ), 1)] t)  (( * ) ((fun y -> y)((fun n -> n) n)) 1)
```

where `t` is the tail of the list :

```
t =  [(Unary (fun x -> x), (fun n -> n - 1), Self)]
```

[3]The tests we ran assert that the behaviour of each generated and programmed CPS function is equal; see § 5 for further discussions regarding a formal result.

Table 5: Automatic generation of Log-Rewind expression evaluation

```
let gen_evalCps = GenCpsState.(genCps
    (fun e -> match e with Prop b -> b | _ -> Nan)
    (fun e _ -> match e with Prop _ -> true | _ -> false)
    (Prop Bot) (fun e -> match e with Prop b -> b | _ -> f_neutral e) f_neutral
    (fun e ->
     match e with
     | And (e1, e2) ->
        [(Binary andT, (fun x -> e1), Self) ;
         (Unary (fun x -> x), (fun x -> e2), Self)]
     | Or (e1, e2) ->
        [(Binary orT, (fun x -> e1), Self) ;
         (Unary (fun x -> x), (fun x -> e2), Self)]
     | Neg e1 ->
        [(Unary negT, (fun x -> e1), Self)]
     | _ -> raise OpenFormula))
```

The function on the left is generated by `gen_f` by considering the multiplication binary operator and the tail of the list: the default argument (that is 1) is used here as a place-holder since the actual value of the first argument of the binary operation (that is the argument of the CPS continuation, say `w`) will be provided by `gen_f`, which will overwrite the second position of the list entry with `w`. The argument on the right evaluates to `n * 1` : note that the second argument of the operation must indeed be its neutral in order to preserve the semantics of the function. The call to `gen_f` with arguments `([(( * ), 1)], t)` produces the value :

```
fun w -> let l = updateR acc ((fun x -> x) w) in
           k ((fun x -> x) n) (gen_f l [])
```

The list `l` is equal to `[(( * ), w)]`, that is the argument `w` substitutes the default value `1`, which is discarded. The call `k ((fun x -> x) n) (gen_f l [])` is the counterpart of the call in line 16 of Table 4: that is, the recursive function `k` of Table 3 represents function `fact` of Table 4. It remains to analyze the CPS continuation passed to `k`. This is the function built by the call `gen_f l []` which returns the value `fun v -> p (app l v)` , which can be unfolded as `fun v -> p (( * ) w v)`, since `app` recursively build an application from a list of entries of type `('a -> 'b -> 'b) * 'a`. Finally, we note that `p` is the CPS continuation passed to `k`.

Support for pattern matching. Table 5 shows a further application, that is the generation of a Log-Rewind CPS evaluator for propositional formulae. To generate an evaluator of expressions, we rely on function `f_neutral` to return the neutral associated to the given constructor (c.f. Table 2): as discussed in § 3, we use a special value **N**, written as `Nan`, to represent the neutral of negation. The first argument provided to `genCps` is a function with type `boolE -> t` that returns the value associated to the base case; this is identified by means of the second and third argument. The fourth argument is a function with type `boolE -> t` that returns the element to be saved in the left entry of a pair in the log. The fifth argument is a function with type `boolE -> t` that provides the neutral that will be used in the top-level evaluation of the expression. The last argument is a function that receives an expression of type `boolE` and returns the list representing the structure of the function to be generated, that is summarized by the following facts: (1) `And`, `Or` are fully recursive binary constructors (as Fibonacci), `Neg` is a recursive unary constructor; (2) the arguments of `And`, `Or` are projected in two entries of the list, while the argument of `Neg` is a projected in a single entry.

To illustrate, consider the formula $e \triangleq \neg(\top \wedge (\bot \vee \neg\top))$ and contrast the code in Table 3 with the call below:

```
gen_evalCps e (fun x l -> StateLogRwd.ret x l)
```

The arguments of the call instantiate the value `(k x f)` in the run (line 26), which in turn comports the execution of the else branch of `k` with the actual parameters: the entry `(Nan, id)` is logged, and `gen_` is executed with argument `f_arg e`, that is `[(Unary negT, (fun _ -> e1), Self)]`, where $e_1 \triangleq \top \wedge (\bot \vee \neg\top)$. The function returns the value `k ((fun _ -> e1) n) (fun v -> p (op v))`, which is better simplified as `k e1 (fun v -> p (op v))`: that is, the structure of the expression is unfolded and `k` can now evaluate the inner expression e_1. In the next round, `gen_` is called with argument `[(Binary andT, (fun _ -> Prop Top), Self) ; (Unary (fun x -> x), (fun _ -> e2), Self)]`, where $e_2 = \bot \vee \neg\top$, and returns the value `k (Prop Top) (gen_f [(andT, Nan)] t)`, where `t` is the tail of the list and `Nan` is a place-holder that will be overwritten. Differently from the negation case, the evaluation of e_1 requires an inner step, as $\wedge$ is a binary operator: this is provided by function `gen_f`, that builds the continuation given the accumulator `[(andT, Nan)]` and the tail `t`. The procedure continues with e_2 giving rise to further rounds; finally, the generation ends by returning an evaluator of e.

5 DISCUSSION

This report presents a functional programming pearl that shows how to add trace and step backward and forward functionalities to CPS functions; to achieve maximum generality, the mechanism is deployed by means of a monad transformer. We study some example of the transformation applied to a State monad featuring references and exceptions, and conclude by presenting a functor that allows to automatically generate CPS transformations with trace support given the abstract structure of the recursion.

We envision at least two applications of our construction : support to foundational computing courses in high level education by allowing step-wise execution of algorithms, and support to the software development process. In that direction, we plan several improvements to the actual setting, among which the most relevant are: add interactions (and non-determinism) to the Log-Rewind monad transformer – a first step towards this direction would be to consider a log as a tree (c.f. [4]); add some parametricity to the log entries (c.f. [9]); study the transformation of more monads (e.g. concurrency); provide a mechanized proof of soundness of the automatic generation of CPS transformations – our idea is to use Why3 [5], while we need to asses the platform's support for monadic computations; embed OCaml compilation with automatic generation of Log-Rewind CPS transformations by means of PPX syntactic extensions (ongoing work, http://ctp.di.fct.unl.pt/FACTOR).

REFERENCES

[1] Andrew Appel. 2006. *Compiling with Continuations.* Cambridge University Press.
[2] Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Math. Struct. Comp. Sci.* 2, 4 (1992), 361–391.
[3] Zaynah Dargaye and Xavier Leroy. 2007. Mechanized Verification of CPS Transformations. In *LPAR (LNCS)*, Vol. 4790. Springer, 211–225.
[4] James Koppel et al. 2018. Capturing the future by replaying the past (functional pearl). *PACMPL* 2, ICFP (2018), 76:1–76:29.
[5] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *ESOP (LNCS)*, Vol. 7792. Springer, 125–128.
[6] Xavier Leroy. 2016. Functional programming and type systems. https://xavierleroy.org/mpri/2-4.
[7] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL*. ACM Press, 333–343.

[8] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55 – 92.
[9] Maciej Piróg and Jeremy Gibbons. 2012. Tracing monadic computations and representing effects. In *MSFP (EPTCS)*, Vol. 76. 90–111.
[10] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159.
[11] Peter Thiemann. 2006. WASH Server Pages. In *FLOPS (LNCS)*, Vol. 3945. 277–293.
[12] Philip Wadler. 1995. Monads for Functional Programming. In *AFP (LNCS)*, Vol. 925. Springer, 24–52.

A AUXILIARY CODE

This appendix contains an example of application of the Log-Rewind monad transformer to binary search trees featuring references, exceptions, and a delete operation outlined in § 3, and the full code of the functor GenerateLogRwdCps introduced in § 4.

The code that implements the Log-Rewind CPS delete of an element in a binary search tree is below. The monad transformer `ExceptionTransf` adds `raise` and `trywith` values to monads with the type signature in Table 1 (c.f. [1, 6]).

```
(* Log-Rewind transformation of state monad plus exceptions *)
(* Example of application : binary search trees featuring delete *)
(* The code of delete is inspired by www.dicosmo.org/share/Flyer-OCamlMOOC.pdf *)
open State
open ExceptionTransf
open LogRwdTransf

(* State monad + exceptions *)
module StateException = struct
    include ExceptionTransf(State)
    let ref x = lift (State.ref x)
    let deref r = lift (State.deref r)
    let assign r x = lift (State.assign r x)
  end

(* State + Log-Rewind monad *)
module LogStateException = struct
    include LogRwdTransf(StateException)
    let raiseE ex = lift (StateException.raiseE ex)
    let trywith x f = lift (StateException.trywith x f)
    let ref x = lift (StateException.ref x)
    let deref r = lift (StateException.deref r)
    let assign r x = lift (StateException.assign r x)
  end

type 'a bst = Empty | Node of 'a bst * 'a * 'a bst

let rec showTree p fmt = function
  | Empty -> Format.fprintf fmt "()"
  | Node (l, y, r) ->
     Format.fprintf fmt "@[ ( %a %a@ %a ) @]" p y (showTree p) l (showTree p) r

let rec find_max = function
  | Empty -> assert false
  | Node (_, x, Empty) -> x
  | Node (_, x, r) -> find_max r

exception Not_found of string

(* State-Log-Rewind delete of the entry el in the tree t *)
(* with monadic continuation f and formatter s *)
let log_delete el t f s =
  LogStateException.
  (run (ref el >>= fun rf ->
        (let rec delete = fun x ->
           let rec delete_ t k =
             log (t, k) >>= fun _ ->
             match t with
             | Empty ->
                deref rf >>= fun el ->
                raiseE (Not_found (Format.sprintf "Closest element: %a" s el))
             | Node (l, y, r) ->
                assign rf y >>= fun _ ->
                if x = y then join l r k
                else
                  if x < y
                  then delete_ l (fun lw -> k (Node (lw, y, r)))
                  else delete_ r (fun rw -> k (Node (l, y, rw)))
           and join l r k = match l, r with
             | Empty, r ->
                log (r, k) >>= fun _ -> k r
             | l, Empty ->
                log (l, k) >>= fun _ -> k l
             | l, r ->
                let m = find_max l in
                (delete m) l (fun lw -> k (Node (lw, m, r)))
           in delete_
         in delete el t f)))

(* Testing *)
let test el t p s =
  Format.printf "Deleting key :%a:\n" p el ;
  let g = fun x l -> LogStateException.ret x l in
  let res = log_delete el t g s in
```

```
    let rm = ref res in
    let k = LogStateException.(sizeOfLog !rm) in
    let rn = ref (- 1) in
    Format.printf
      "Going forward from the first to the last call
       of delete with arg %a\n"(showTree p) t;
    for i = 0 to k - 1 do
      let m, r, n = LogStateException.(forward !rm !rn) in
      Format.printf
        "\nPosition : %d\nMonad value : %a\n" (!rn + 1) (showTree p) r ;
      rn := n; rm := m
    done  ;
    Format.printf
      "Resulting bst: %a@."(showTree p) (fst res)

let t =
  let l = Node ( (Node(Empty, 2, Empty), 3, (Node(Empty, 4, Empty)))) in
  let rr = Node(Node(Empty, 11, Empty), 12, (Node(Empty, 13, Empty))) in
  let r = Node ( (Node(Empty, 7, Empty), 9, (Node(Empty, 10, rr)))) in
  Node (l, 5, r)

let () =
  let f = fun () -> string_of_int in
  test 9 t Format.pp_print_int f;
  test 15 t Format.pp_print_int f
```

The code of the functor to generate Log-Rewind CPS evaluators is below.

```
open LogRwdTransf

module GenerateLogRwdCps(M :  MONAD) = struct
    module LogRwdMonad = struct
        include LogRwdTransf(M)
      end

    type ('a, 'b) un = 'a -> 'b
    type ('a, 'b) bin = 'a -> 'a -> 'b
    type ('a, 'b) func =
      | Unary of ('a, 'b) un
      | Binary of ('a, 'b) bin
      | Self

    exception ArityMismatch of string
    exception SemanticsMismatch of string
    exception EmptyList of string

    let genCps f_base pred_base tree_shold f_ind f_neutral f_args =
      let rec app l x
        = match l with
        | (f, v) :: [] -> f v x
        | (f, v) :: t -> f v (app t x)
        | _ -> raise (EmptyList __LOC__) in
      let updateR l v =
        match List.rev l with
        | h :: t ->
           (match h with (op, _) -> List.rev ((op, v) :: t))
        | _ -> raise (EmptyList __LOC__) in
      fun x f ->
      let actual_x = ref x in
      (* neutral of first operation *)
      let default = f_neutral x in
      LogRwdMonad.
      (run
         (let rec k n p  =
            if pred_base n tree_shold
            then
              log (f_base n, p) >>= fun _ ->
              p (f_base n)
            else
              log (f_ind n, p) >>= fun _ ->
              let rec gen_f acc = function
                | [] -> fun v -> p (app acc v)
                | (opf, x, gf) :: []  ->
                   (match opf with
                    | Unary g_b_op ->
                       (match gf with
                        | Unary g ->
                           let l = updateR acc (g_b_op (g (x n)))
                           in gen_f l []
                        | Self ->
                           fun w ->
                           let l = updateR acc (g_b_op w)
                           in (actual_x := x n ;
                               k (x n) (gen_f l []))
                        | Binary _ -> raise (ArityMismatch __LOC__))
                    | _ -> raise (SemanticsMismatch __LOC__) )
                | (opf, x, gf) :: t  ->
                   match opf with
                   | Unary g ->  raise (ArityMismatch __LOC__)
                   | Binary op ->
                      fun w ->
                      let l = updateR acc w in
                      (actual_x := x n ;
                       k (x n) (gen_f (l @ [(op, default)]) t))
                   | _ -> raise (SemanticsMismatch __LOC__)
            in
            let gen_ = function
              | (opf, x, gf) :: t ->
                 (match gf with
                  | Self ->
                     (match t with
                      | [] ->
                         (match opf with
                          | Unary op ->
                             (actual_x := x n ;
                              k (x n) (fun v -> p (op v)))
                          | Self -> raise (SemanticsMismatch __LOC__)
                          | _ -> raise (ArityMismatch __LOC__))
                      | _ ->
                         (match opf with
                          | Unary op -> raise (ArityMismatch __LOC__)
                          | Self -> raise (SemanticsMismatch __LOC__)
                          | Binary op ->
                             (actual_x := x n ; k (x n) (gen_f [(op, default)] t))))
                  | Unary g  ->
                     (match t with
                      | [] ->
                         (match opf with
                          | Unary op -> p (op (g (x n)))
                          | Self -> raise (SemanticsMismatch __LOC__)
                          | _ -> raise (ArityMismatch __LOC__))
                      | _ ->
                         (match opf with
                          | Unary _ ->  raise (ArityMismatch __LOC__)
                          | Self -> raise (SemanticsMismatch __LOC__)
                          | Binary op ->
                             (gen_f [(op, default)] t)  (op (g (x n)) default)))
                  | _ -> raise (ArityMismatch __LOC__))
              | _ -> raise (EmptyList __LOC__)
            in gen_ (f_args !actual_x)
          in k x f))

end
```