Lightweight Linearly-typed Programming with Lenses and Monads

Keigo Imai¹,a)  Jacques Garrigue²,b)

Received: September 26, 2018, Accepted: December 29, 2018

Abstract: This paper shows an encoding of linear types in OCaml and its applications. The encoding enables to write correct OCaml programs based on safe resource access guided by linear types. Linear types ensure that every variable is used exactly once, and, thus, they can be used to check the behavioural aspects of programs such as resource access and communication protocols in a static way. However, linear types require significant effort to be integrated into existing programming languages. Our encoding allows the vanilla OCaml typechecker to enforce linearity by using lenses and a parameterised monad. Parameterised monads are monads with a pre- and a post-condition, and we use them to track the creation and consumption of resources at the type level. Lenses, which point at parts of a data type, are used to refer to a resource in pre- and post-conditions. To handle comfortably structured data such as linearly typed lists, we further propose an extension to pattern matching based on the syntax-extension mechanism of OCaml. We show an application to static checking of communication protocols in OCaml.

Keywords: OCaml, linear types, functional programming, monad, lens

1. Introduction

Linear types guarantee that their values are used only once; thus, they have been utilised to statically analyse the behaviour of programs, such as resource usage and communication protocol.

Several programming languages have adopted linear-like types, such as affine types in Rust [26] and uniqueness types in Clean [23]. In Rust, affine types are exploited to statically track variable occurrences, achieving efficient memory management. A recent proposal [4] also introduces linear types into the Glasgow Haskell Compiler [16].

However, introducing linear types into an existing programming language incurs a high implementation cost, as it requires modifying its compiler and type system. If one could encode them through a library using only built-in language features, in a customizable way, it would not only allow linearly typed resource-safe programming in that language, but also encourage experimenting with various programming techniques based on the combination of linear types and other language features.

Embedding linear types in Haskell. Techniques to encode linear types have been developed for Haskell. In particular, Polakow [24] directly embeds the linear lambda calculus in higher order abstract syntax (HOAS) [20] form, making it readily usable. However, his embedding relies on Haskell type classes and functional dependencies [12] to track the consumption of linear values; thus, it is difficult to adapt it to other programming languages.

In this paper, we show an embedding of linear types in OCaml using a parameterised monad [1] and lenses [5], [21], and propose as library linocaml built around it. A parameterised monad is a monad with extra type parameters representing the pre- and post-conditions of monadic computations, which statically encode the generation and consumption of linear resources. A lens is a functional reference that points to a position in a data type and is used as a reference to a linear resource in a pre- or post-condition.

The linear type encoding presented in this paper depends only on parametric polymorphism, available in many programming languages, thus achieving a lightweight and portable implementation.

We also provide extra features for pattern matching against structured data, such as linearly typed lists, by using the syntax extension mechanism of OCaml, thus allowing much more flexible programming with linear types. As an application of our pattern matching extension, we introduce an encoding of session types [8] and show a solution to the so-called Santa Claus problem in concurrent programming [2], [27].

Prior work by the authors. This paper builds on Garrigue’s Safeio API [6], and on the encoding of session types proposed by Imai et al. [9], [10]. Both used some form of parameterised monads and lenses. Pattern matching for linear types was also used in Ref. [10], albeit in a limited way.

The contribution of this paper is to extend the linear type encoding in Safeio and our prior session type work, so as to allow linear functional programming on structured data such as arrays.
and lists.

Structure of this paper. The rest of this paper is organised as follows: In Section 2, we introduce an example of programming with linear types. Section 3 shows our linear type encoding using a parameterised monad and lenses in OCaml. In Section 4 we introduce a syntax extension to handle linearly typed structured data. As an application, we show a solution to the Santa Claus problem using a session type encoding and linearly typed structured data in Section 5. Section 6 introduces related work, and Section 7 concludes this paper.

Source code. linocaml is available at https://github.com/keigo/linocaml.

2. Linear Types and Resource Control

We overview programming with linear types using Wadler’s example [28]. In this section, we consider a purely functional API for linear arrays in an imaginary programming language equipped with linear types. In the following, we use the OCaml syntax, i.e. type variables are annotated with quotes (‘a,’ b, ... ) and type constructors are written in a postfix manner (’a list for a list of type ’a).

Example 2.1 (Linearly typed array API) We give a linearly typed array API as follows, where ’a larr is the type of linearly typed arrays whose element type is ’a.

```ocaml
let alloc : ’a list -> ’a larr
val dealloc : ’a larr -> unit
val lookup : int -> ’a larr -> ’a larr * ’a
val update : int -> ’a larr -> ’a larr * ’a
val map : ’(’a -> ’b) -> ’a larr -> ’b larr
val to_list : ’a larr -> ’a larr * ’a list
```

The usage of each operation is as follows:

- alloc xs creates an array from the elements in list xs.
- dealloc arr deallocates array arr.
- lookup i arr returns the pair of arr itself and the i-th element of array arr (array bounds are not checked statically, but at run-time).
- update i e arr returns an array with i-th element e and all other elements having the same value as in arr.
- map f arr returns an array whose elements are the result of applying f to each element in arr.
- to_list arr returns the pair of arr itself and the list of the elements of arr.

The linearly typed array API enables one to reuse its memory location after use, and, in particular, it allows in-place update of types of elements in an array.

Example 2.2 (Type updating) The following shows an example of updating types of elements in an array.

```ocaml
let arr = alloc [100; 200; 300] in
let arr1 = map string_of_int arr in
let arr2 = update 1 "Hello" arr1 in
let arr3, x = lookup 1 arr2 in
dealloc arr3;
print_endline x
```

This code initially allocates an array of type int larr containing 100, 200, 300, and then it converts (map) it into an array of strings (type string larr) by applying string_of_int to each element. Then, it updates the first element with “Hello”, looks up that element and binds it to x to finally print it after deallocating the array. Linear typing allows us to ensure that the array variables arr, arr1, arr2 and arr3 are consumed exactly once. Therefore, purely functional operations such as map and update can be implemented by in-place (destructive) memory update.

A violation of linearity may lead to unsafe behaviour. For example, the following code is unsafe:

```ocaml
let arr = alloc [100; 200; 300] in
let arr1 = map string_of_int arr in
update 1 400 arr
```

It first allocates an array of integers arr and then converts it to a string array arr1. Here, the third line is unsafe because it writes an integer 400 into the string array. A linear type system can preclude such violations statically.

3. Encoding Linear Types Using a Parameterised Monad and Lenses

Variable bindings like arr, arr1, ... in Example 2.2 may violate linearity since the OCaml type system does not track the number of occurrences of a variable. From this observation, we developed a combinator library LinMonad based on parameterised monads, which provides a way to implicitly handle linear values without any variable binding. The uses of linear values appear explicitly in the monad type; thus, the linearity constraints can be statically guaranteed by the OCaml type system.

First, in Section 3.1, we introduce a framework to statically track the generation and consumption of a single linear resource. Next, in Section 3.2, we develop a framework that manipulates multiple linear resources by having the pre- and post-conditions hold slot sequences, and by introducing lenses that refer to individual linear resources in these sequences. Section 3.3 shows an implementation of LinMonad based on the state monad, and introduces a technique to implement specific linearly typed APIs in this framework.

3.1 A Parameterised Monad

For an example of programming using LinMonad, in Fig. 1, we show an API for linear arrays that encodes the linearly typed functions of Example 2.1. Each function returns a monadic value (command) that represents an array operation, rather than an array. A monadic value has type (pre, post, α) monad, and two monadic values can be concatenated like a UNIX command to make a compound one. The type pre is an input value from the previous command, post is an output to the next command, and α is the result of the computation.

Type ’a larr is declared as an alias for ’a array lin, where lin is the type constructor that distinguishes linear types. The type of the computation result is wrapped with data, representing an unrestricted (non-linear) type. Constructors lin and data do not have a special role in this section; however, they play a

□

22 However, since arrays of floating-point numbers (float array) in OCaml are specialised (unboxed) [15], their memory representation is not compatible with that of other arrays, and special care will be required in the implementation (see Section 3.3).
type 'a larr = 'a array lin
val alloc : 'a list ->
  (empty, 'a larr, unit data) monad
val dealloc :
  ('a larr, empty, unit data) monad
val lookup : int ->
  ('a larr, 'a larr, 'a data) monad
val update : int -> 'a ->
  ('a larr, 'a larr, 'a data) monad
val map : ('a -> 'b) ->
  ('a larr, 'b larr, unit data) monad
val to_list :
  ('a larr, 'a larr, 'a list data) monad

Fig. 1 A linearly typed array API based on LinMonad.

The computation result of a monadic value can be bound to the parameter of the function like in $e_1 >>\ fun \ x -> e_2$, and can be used in the subsequent computation\(^{14}\). An expression of the form $e_1 >> e_2$ has almost the same behaviour as the bind operation, except that it discards the result of the lhs. It could be written as $let \ m = e_2 \ in \ e_1 >>> \ fun \ _ -> m\(^{15}\).

The other property of linearity, which stipulates that linear values are not duplicated, is guaranteed by the fact that the inputs and outputs of commands are implicitly threaded by the bind operation and are never bound to a variable.

The empty value is represented by the constructor Empty. The commands are executed via the function run. The output type empty in run ensures that the last command does not output a linear value.

Example 3.1 (Array operations using a monad) The following program simulates Example 2.2 using the parameterised monad.

```ocaml
val ex1: unit -> (empty, empty, unit data) monad
val ex2: unit -> (empty, empty, unit data) monad
let ex1 () =
  alloc [100; 200; 300] >>
  string_of_int >>>
  alloc [100; 200; 300] >>
  map (fun x ->
    print_endline x;
    return ());
let () = run ex1
```

In this example, the array generated by alloc is manipulated using map, update, and lookup, and then destructed by dealloc. Overall, function ex1 returns a command with empty input and output that operates on an array, as shown in Example 2.2.\(^{16}\)

3.2 Lenses Focusing on Multiple Linear Resources

We introduce a framework to handle multiple linear values simultaneously in LinMonad. For example, it allows to write operations analogous to the following:

```ocaml
let arr1', x1 = lookup i arr1 in
let arr2', x2 = lookup i arr2 in
x1 + x2
```

which compute the sum of the i-th elements of two linearly typed arrays. The idea is to use a data structure called slot sequence\[^{6}\], \[^{9}\], \[^{10}\], which holds multiple linear resources, in the input and output of commands in the parameterised monad. Lenses enable one to indirectly refer to linear resources while restricting non-linear access to them as before, thereby guaranteeing linearity.

\(^{14}\) The operator precedence is as follows: $e_1 >>> (fun \ x -> e_2)$.
\(^{15}\) The let-binding is required because OCaml is not pure, and the expression $e_2$ may have side effects.
\(^{16}\) Note that (print_endline x; return ()) discards the unit value () returned by print_endline at the lhs of $;$ and returns a pure command that does nothing.
3.2.1 Preliminaries

Slot sequences. A slot sequence holds multiple linear resources. It is a data structure composed of pairs nested on the right \((x_0,(x_1,(x_2,\ldots)))\). Although at any point there may only be a finite number of linear resources available, it is useful to be able to assume the existence of an infinite number of slots. In particular, one can denote the absence of any linear resource by an infinite sequence \(\text{empty} * (\text{empty} * (\text{empty} * \ldots))\). There are two ways in Ocaml to write types with such an infinite structure. (1) Ocaml’s polymorphic variants or objects allow equi-recursive types, but they can be mimicked using finite unrolling (Section 3.2.3). (2) The -rectypes option of the Ocaml compiler enables to construct equi-recursive types through arbitrary type constructors so that \(\text{empty} * (\text{empty} * \ldots)\) can be written more directly as \(\text{empty} * 't \equiv 't\).

In this paper, we use the latter for simplicity, but our library actually uses the former. More generally, most programming languages do not provide equi-recursive types, but they can be mimicked using finite unrolling (Section 3.2.3).

Lenses. A lens [5] is an abstraction of bi-directional transformation in the context of bidirectional programming. It consists of a transformation from one data structure (source) to another data structure (view) and a backward transformation of the changes made in the view into the source. In addition, Haskell’s view pattern and Pickering’s lenses [21] allow writing back values made in the view into the source. In addition, Haskell’s structure (view) and a backward transformation of the changes picked using finite unrolling (Section 3.2.3).

Example 3.2 (Manipulating slots using lenses) The following example shows the intuitive behaviour of lenses focusing on slot sequences. Let Empty be the constructor of type empty. Then, the infinite slot sequence of Empty which has type empty * 't as 't is defined as follows:

\[
\begin{align*}
\text{val all_emp} & \colon \text{empty} * 't \equiv 't \\
\text{let rec all_emp} & = \text{Empty}, \text{all_emp}
\end{align*}
\]

This is a cyclic list having Empty as its head. The following code uses \_0 to assign an array \(\text{arr} : \text{int} \ \text{larr}\) to the 0-th position of the empty slot sequence:

\[
\begin{align*}
\text{val slots1} & \colon \text{int} \ \text{larr} \ : \ (\text{empty} * 't \equiv 't) \\
\text{let slots1} & = \_0.\text{put all_emp} \ \text{arr}
\end{align*}
\]

The composite lens \(\text{succ} (\text{succ} \_0)\) in the following code assigns an array \(\text{arr} : \text{int} \ \text{larr}\) to the second position in the sequence.

\[
\begin{align*}
\text{val slots2} & : \text{int} \ \text{larr} \ : \ (\text{empty} * (\text{string} \ \text{larr}) \equiv (\text{empty} * 't \equiv 't)) \\
\text{let slots2} & = (\text{succ} (\text{succ} \_0)).\text{put slots1} \ \text{arr}
\end{align*}
\]

3.2.2 Lenses manipulating linear values in the monad

In Fig. 4, we present the type of slot sequences, the function \run\(\) executing the command, and the operator \(\oplus\) which updates an element in the slot sequence using a command. Expression \(m \oplus 1\) executes the computation \(m\) in the slot referenced by lens 1. The type signature reads as follows:

- From the pre-condition ‘pre of \(m \oplus 1\), a linear resource of type ‘p is obtained using lens 1.
- The resource ‘p is consumed in the monadic computation m, and the post-condition ‘q is produced and the result value ‘a is returned.

\[
\begin{align*}
\text{val all_empty} & = \text{empty} * 't \equiv 't \\
\text{val run} & : (\text{unit} \rightarrow (\text{all_empty}, \text{all_empty}, 'a \ \text{data}) \ \text{monad}) \\
& \rightarrow 'a \\
\text{val @\oplus} & : ( 'p, 'q, 'a) \ \text{monad} \\
& \rightarrow ( 'p, 'q, 'pre, 'post) \ \text{monad} \\
& \rightarrow ( 'pre, 'post, 'a) \ \text{monad}
\end{align*}
\]

Fig. 4 An operator for slot update in the LinMonad.

\[\text{\footnotesize{(3) We count the first element of the slot sequence as the 0-th slot.}}\]
Again by lens \( l \), 'q is written back to 'post and it becomes the post-condition of \( n \oplus l \).

The following example handles multiple linear resources using lenses. We introduce the functions \( \text{iteriM} \) and \( \text{mapiM} \) as variants of \text{List.iter} and \text{List.map} in OCaml, respectively. For instance, \( \text{iteriM} f l \) executes the command \( f \ i \ e_i \) for each element of \( l \), where \( i \) is the position of \( e_i \) in \( l \).

**Example 3.3 (Handling multiple linear resources (1))** To show an example of accessing multiple resources, we consider calculating the sum of two arrays. The following function returns an array \([123, 234, 345]\) by calculating the sum of each element of the same index in two arrays: \([23, 34, 45] \) and \([100, 200, 300]\), respectively.

```
val ex2 : unit ->
  (all_empty, all_empty, int list data) monad
let ex2 () =
  alloc [23; 34; 45] @> _0
  alloc [100; 200; 300] @> _1
  iteriM (fun i x ->
    lookup i @> _1 >>= fun y ->
      update i (x + y) @> _1)
  to_list @> _0 >>= fun xs ->
  return xs
```

By using the @> operator, we can assign the two newly allocated arrays to the 0-th and first slots, respectively. Then, the function \( \text{iteriM} \) is used to update the first array with the sum of each element at index \( i \) from the two arrays, and, finally, the first array is converted to a list. The anonymous function \( \text{fun} \ i \ x \rightarrow \ldots \) passed to \( \text{iteriM} \) reads the \( i \)-th element of the first array by calling \( \text{lookup} \ i \ @> _1 \) and writes back the sum by calling update \( i \ (x + y) @> _1 \).

Next is an example that updates the type of array elements.

**Example 3.4 (Handling multiple linear resources (2))** The following program converts the array \([100, 200, 300]\) to an array of strings and then concatenates "abc", "def", "ghi" to each element to obtain the array "abc123", "def234", "ghi345".

```
val ex3 : unit ->
  (all_empty, all_empty, string list data) monad
let ex3 () =
  alloc [100; 200; 300] @> _0
  alloc ["abc"; "def"; "ghi"] @> _1
  mapiM (fun i x ->
    lookup i @> _1 >>= fun s ->
      return (s ^ string_of_int x)) _0
  to_list @> _0 >>= fun xs ->
  dealloc @> _0
  dealloc @> _1
  return xs
```

Figure 5 shows the type signatures of \( \text{iteriM} \) and \( \text{mapiM} \).

```
val iteriM :
  (int -> 'a -> ('pre, 'pre, unit data) monad)
-> ('a larr, 'a larr, 'pre, 'pre) lens
-> ('pre, 'pre, unit data) monad
val mapiM :
  (int -> 'a -> ('pre, 'pre, 'b data) monad)
-> ('a larr, 'b larr, 'pre, 'post) lens
-> ('pre, 'pre, unit data) monad
Fig. 5 Signatures for \text{iteriM} and \text{mapiM}.
```

which represent the following linearity constraints:

- In \( \text{iteriM} f l \), it is not possible for the first argument \( f \) to update the type of the array elements since it is called many times during the iteration. For this reason, the pre- and post-conditions of the type of the monadic value are both \( 'pre ^8 \).
- The second argument \( l \) is a lens referring to an array to be iterated. Since \( \text{iteriM} \) does not update the type, the third and fourth type arguments of the lens are identical to the first type argument \( 'a \) larr and to the second type argument \( 'pre \), respectively.

- The first argument \( f \) of \( \text{mapiM} f l \) does not update the type as well, but it returns the converted element \( 'b \) data as the result value of the monad. On the other hand, the second argument \( l \) referring to the array indicates that this function updates the value of the linear type \( 'a \) larr and returns a new array \( 'b \) larr. Reflecting this update, the post-condition type of the command is \( 'post \), which is the updated source type by the lens.

### 3.2.3 Typing Slots Without Equi-recursive Types

We show how the slot sequences can be represented in programming languages without equi-recursive types. For this, we introduce the functions \( \text{extend} \) and \( \text{shrink} \) which expand or shrink the slot sequence by one, respectively.

```
val extend : ('pre, empty * 'pre, unit data) monad
val shrink : (empty * 'pre, 'pre, unit data) monad
```

Functions \( \text{extend} \) and \( \text{shrink} \) enable to handle as many slots as required.

**Example 3.5 (Expanding/shrinking of a sequence)** Function \( \text{example3} \) in Example 3.4 can be typed and executed without any infinite slot sequence by using \( \text{extend} \) to expand the slot sequence by two. Since the output of \( \text{run} \) must be empty, the slot sequence is shrunk at the end using \( \text{shrink} \).

```
val ex4 : unit ->
  (empty, empty, string list data) monad
let ex4 () =
  extend >>
  example3 () >>= fun x ->
  shrink >>
  shrink >>
  return x
let () = run ex4
```

### 3.2.4 Polymorphic Lenses Using GADTs

As mentioned in Section 3.2.1, lenses like \( \text{succ} \) are subject to the value restriction and cannot have a polymorphic type. For this reason, to manipulate data of different types, we need to locally combine lenses like \( \text{succ} \) (\( \text{succ} \_0 \)), which is cumbersome.

Lenses defined using generalised algebraic data types (GADTs)\(^8\) constructors as in Figure 6 can avoid the value restriction, keeping such combinations polymorphic.

\( \text{Fst} \) is a lens that refers to the 0-th slot, and \( \text{Next} \ l \) is a lens.

\( ^8 \)Unlike with function return, the command returned by \( f \) may include side effects. The type signature only states that \( f \) does not update the types.
that refers to the next element of \( l \) which is equivalent to \( \text{succ} \ l \).

\( \text{Any} \ (\text{get}, \text{put}) \) is a lens consisting of an arbitrary view function \( \text{get} \) and putback function \( \text{put} \).

3.3 Implementations of the Monad and APIs

\textbf{LinMonad as a state monad.} Figure 7 shows an implementation of LinMonad (Fig. 2) and slot-based operations based on the state monad [29]. The type \( (\text{pre}, \text{post}, \text{a}) \) monad denotes a state monad with state changing from \( \text{pre} \) to \( \text{post} \), implemented as the function type \( \text{pre} \rightarrow \text{post} \rightarrow \text{a} \).

Monadic operations \( \text{return} \), \( \text{bind} \), etc. are standard; however, they wrap the result value with a \text{Data} constructor. The function run explicitly handles the state value \( \text{Empty} \) to ensure that the pre- and post-conditions will be empty. Here, \text{run} is almost the same except that it uses \text{all_emp} instead of \text{Empty}.

\textbf{Implementing a linearly typed API} When implementing a linearly typed API, one works “under the hood”, using the vanilla OCaml type system, which doesn’t ensure or exploit linearity. This requires techniques specialised to the domains and properties to be guaranteed. For example, in an efficient array implementation, unsafe operations are required to implement updates that change the type of the array.

Figure 8 shows an implementation of linearly typed arrays with type update. The implementation employs a few tricks to work around the specialised (unboxed) memory representation that OCaml uses for floating-point numbers [15]. The following type-unsafe functions are used to encode and decode array elements, and to convert the type of arrays.

\begin{verbatim}
val Obj.repr : 'a -> Obj.t
val Obj.obj : Obj.t -> 'a
val Obj.magic : 'a -> 'b

let alloc xs = fun Empty ->
  let arr = Array.make (List.length l) (Obj.repr @)
  in
  List.iteri (fun i a ->
    arr.(i) <- (Obj.repr a)) xs;
  Lin_arr, Data ()

let dealloc arr = Empty, ()

let lookup i = fun ((Lin_arr) as pre) ->
  pre, (Obj.obj arr.(i))

let update i a = fun ((Lin_arr) as pre) ->
  arr.(i) <- (Obj.repr a);
  pre, ()

let map f = fun (Lin_arr) as pre ->
  Array.iteri (fun i a ->
    arr.(i) <- (Obj.repr (f (Obj.obj a))))
  arr;
  Lin_arr, Data ()

let to_list = fun ((Lin_arr) as pre) ->
  pre, (Obj.magic (Array.to_list arr))

Fig. 8 An implementation of linearly typed arrays.
\end{verbatim}
4. A Pattern Matching Extension

In this section, we show a syntax extension of pattern matching on linearly typed structured data. Pattern matching is a powerful feature of functional programming with which one can express a variety of algorithms when used in combination with recursive data structures. In particular, linearly typed lists are important because they allow to dynamically handle an arbitrary number of linear resources.

4.1 Linearly Typed Patterns and a Revised Array API

To enable pattern matching against linearly typed values, we extend the type of the result of the parameterised monad to include linear types as in ('pre', 'post', 'a lin monad).

In addition, we introduce a syntax extension let %lin for pattern matching against such linear result values. The expression let %lin pat = e₁ in e₂ executes e₁, binds its result to pattern pat, and then executes e₂. Pattern pat is extended to include lens pattern #l, which assigns the matched linear value into an empty slot referred to by #l.

Here, we show an example of array manipulation using lens patterns. To compare programming with lens patterns with the style of Section 3, we first introduce a new array API with linear types as in Fig. 2, which directly handles linear values.

Example 4.1 (A revised array API) Figure 9 introduces a revised version of the array API. We summarise the changes from Fig. 1 in Section 3 as follows:

- Each function originally returning an array as output type now has a return type containing a linear array.
- Functions other than alloca take as a parameter a lens of type (‘a larr, empty, ‘pre, ‘post) lens, which reflects the fact that they consume an array referred to by this lens. In other words, they consume the array ‘a larr in slot sequence ‘pre and then the slot is emptied and written back to ‘post. The return type is the (‘pre, ‘post, t) monad, where the pre-condition ‘pre and the post-condition ‘post are the same as those manipulated by the lens.
- Unlimited (non-linear) types ‘a are wrapped as ‘a data to separate them from linear types. For example, the result of lookup is a pair type (‘a larr * ‘a data) lin of a linear array and the element found in that array.

Programming with lens patterns closely resembles that in Section 2, which directly handles linear values.

Example 4.2 (Lens patterns matched against arrays) Using lens patterns and the array API in Fig. 9, we can write the array operation of Example 2.2 as follows:

```ocaml
val alloc : 'a list -> ('pre, 'pre, 'a larr) monad
let alloc xs = fun pre ->
  pre, Lin_. (Array.of_list pre)
val dealloc : ('a larr, empty, ('pre, 'post) lens ->
  ('pre, 'post, unit data) monad
let dealloc 1 = fun pre ->
  l.put post Empty, Data ()
val lookup : int ->
  ('a larr, empty, ('pre, 'post) lens ->
  ('pre, 'post, ('a larr * 'a data) lin) monad
let lookup i l = fun pre ->
  let ((Lin_. arr) as arr) = l.get pre in
  l.put pre Empty, Lin_. (Array.map f arr)
```

It can be seen that the only difference from Example 2.2 above is the extension point %lin and the symbol # for lens patterns.

Multiple arrays can also be intuitively manipulated by lens patterns.

Example 4.3 (Handling multiple arrays with lens patterns) Using lens patterns, we can write the program of Example 3.4 as follows:

```ocaml
val alloc : 'a list -> ('pre, 'pre, 'a larr) monad
let alloc xs = fun pre ->
  pre, Lin_. (Array.of_list pre)
val dealloc : ('a larr, empty, ('pre, 'post) lens ->
  ('pre, 'post, unit data) monad
let dealloc 1 = fun pre ->
  l.put post Empty, Data ()
val lookup : int ->
  ('a larr, empty, ('pre, 'post) lens ->
  ('pre, 'post, ('a larr * 'a data) lin) monad
let lookup i l = fun pre ->
  let ((Lin_. arr) as arr) = l.get pre in
  l.put pre Empty, Lin_. (Array.map f arr)
```

It can be seen that the only difference from Example 2.2 above is the extension point %lin and the symbol # for lens patterns.

Multiple arrays can also be intuitively manipulated by lens patterns.
expansion. First, let us consider the case where \(\text{\texttt{\lin}}\) contains a lens pattern only and generalise it for structured patterns.

Expressions with lens patterns

\[
\text{\texttt{\let\lin \#1 = e_1 \ \text{in} \ e_2}}
\]

are treated as an abbreviation of the following expression:

\[
e_1 \Rightarrow \scripts{\text{\texttt{\fun\lin \#1} \Rightarrow e_2}}
\]

Here, \(\Rightarrow\) is a variant of the bind function, which requires a \(\text{\texttt{\fun\lin}}\) function on its right-hand side.

The function \(\text{\texttt{\fun\lin \#1} \Rightarrow e_2}\) is expanded to

\[
\text{\texttt{\bind\# (fun tmp -> _put l tmp \Rightarrow e_2)}}
\]

by the preprocessor, where \(tmp\) is a fresh variable. \(\text{\texttt{\bind\#}}\) is a constructor that distinguishes a \(\text{\texttt{\fun\lin}}\) function at the type level and should not be used by programmers. By this, the rhs of bind is statically enforced to be a \(\text{\texttt{\lin}}\) function. The expression \_put \_l \(v\) is a command to store the value \(v\) in the slot pointed to by lens \(l\). The type of the syntax extension \(\text{\texttt{\lin}}\) has the following form:

\[
(\text{\texttt{\fun\lin \#1} \Rightarrow e_2}) := (\alpha \ \text{lin} \Rightarrow \pre\ \text{post}, \beta \ \text{monad}) \ \text{bind}
\]

where \(\alpha\), \(\pre\), \(\post\), and \(\beta\) are determined by the type of lens \(l\) and by expression \(e_2\). For example, an expression with lens \_0 \texttt{\_l} would have the following type:

\[
(\text{\texttt{\fun\lin \_0} \Rightarrow \text{\texttt{return}}}) := (\alpha \ \text{lin} \Rightarrow \pre\ \text{post}, \beta \ \text{unit data}) \ \text{bind}
\]

This function takes a linear value and returns a command that stores it in the 0-th slot.

As a whole, the syntax extension \(\text{\texttt{\let\lin \#1 = e_1 \ \text{in} \ e_2}}\) is expanded to \(e_1 \Rightarrow \text{\texttt{\bind\# (fun tmp -> _put l tmp \Rightarrow e_2)}}\). Because the variable \(tmp\) bound to the linear result of \(e_1\) is immediately assigned to slot \(l\), and because \(tmp\) does not occur anywhere else, linearity is maintained.

**Figure 10** shows the signature and implementation of the auxiliary functions used by \(\text{\texttt{\lin}}\). Operator \(\Rightarrow\) is a specialised bind function that takes a function wrapped by \(\text{\texttt{\bind\#}}\) on its rhs. Function \_put outputs a modified slot sequence that stores a linear value in the input pre.

**Linear and unlimited variable patterns.** We introduce variable patterns to handle mixed linear and unlimited values, such as the result of \texttt{lookup}. Special care is needed for variable patterns as there is a risk that a linear value of type \(\text{\texttt{\lin}}\) might be bound to a variable pattern that has a polymorphic type. Here, we assume that the top level of the pattern of \(\text{\texttt{\lin}}\) is a linear value and that the unlimited value appearing within it is wrapped with \(\text{Data}\). We also assume that no linear value appears inside an unlimited value. Under these assumptions, the preprocessor checks that variable patterns never appear at the top level and that they are wrapped by a \(\text{Data}\) constructor, and if these conditions are violated, the preprocessor will report a syntax error, ensuring that variable patterns do not bind linear values.

**Omission of Data in variable patterns.** Because it is cumbersome to always wrap a variable pattern with \(\text{Data}\), we use the following rules for pattern expansion:

- Variable patterns are allowed inside the \(\text{Data}\) constructor, like \texttt{data x}.
- Variable patterns outside \(\text{Data}\) are implicitly wrapped by \(\text{Data}\) to have type \(\tau\ \text{data}\).

This makes it possible to omit the \(\text{Data}\) constructor, e.g., the pattern \(\text{\#arr(x)}\) binds the result of \texttt{lookup} which is of type \(\text{\#a larr * \#a data}\), while structured pattern matching for unlimited values like \(\\#arr, \text{Data (Some x)}\), which is of type \(\text{\#a larr * \#a option data}\), becomes possible by writing \(\text{Data}\) explicitly.

We further introduce syntax extensions \(\text{\texttt{\match\lin}}\) and \(\text{\texttt{\functions\lin}}\), which extend \(\text{\texttt{\match}}\) and \(\text{\texttt{\function}}\) with lens patterns, respectively. We summarise the expansion of each syntax extension as follows:

- \(\text{\texttt{\fun\lin pat} \Rightarrow e}\) expands to
  \[
  \text{\texttt{\bind\# (fun conv(pat) \Rightarrow puts(pat) \Rightarrow e)}}
  \]
  Here, \(\text{\texttt{\conv}}\) is a syntactic function for converting the pattern \(pat\), which is given later. \(\text{\texttt{\puts}}\) is a function that generates a command
  \[
  \text{\texttt{_put l1 tmp1 \Rightarrow ... \Rightarrow _put ln tmpn}}
  \]
  for the lens \(l_1, \ldots, l_n\) appearing in \(pat\), where the variables \(tmp1, \ldots, tmpn\) are freshly generated by \(\text{\texttt{\conv}}\).
- \(\text{\texttt{\function\lin case1, \ldots, case_n}}\) expands each clause \(case_i\) of the form \(\text{\texttt{pat} \Rightarrow e_i}\) as in \(\text{\texttt{\fun\lin}}\).
- \(\text{\texttt{\let\lin pat = e_1 \ \text{in} \ e_2}}\) is expanded similarly to the case of \(e_1 \Rightarrow \text{\texttt{\fun\lin pat e_2}}\).
- \(\text{\texttt{\match\lin e with case1, \ldots, case_n}}\) expands similarly to \(e \Rightarrow \text{\texttt{\function\lin case1, \ldots, case_n}}\).

**Figure 11** shows a pseudo-OCaml code for expanding \(\text{\texttt{\lin}}\) patterns. Function \_traverse converts pattern \(p\) recursively. In the case of a lens pattern, it replaces the pattern with a fresh variable and records the pair of the lens and the variable for later insertion of \_put. On the other hand, in the case of a variable pattern \(x\), it is expanded to pattern \(\text{\texttt{Data x}}\) of type \(\text{\#a \data}\). For a constructor pattern, it converts the argument pattern recursively. An error occurs for patterns leaking linear values, e.g., \texttt{as-patterns}.

**Type safety in user programs.** In summary, by using this library and accompanying syntax extensions, the programmer is guaranteed that values of type \(\text{\texttt{\lin}}\) are used linearly, provided she/he does not directly use the constructors \(\text{\texttt{\Lin\#}}\) and \(\text{\texttt{\bind\#}}\). Specifically, linear values are always stored in the slot sequence in the monad, and although one can create a function that takes a parameter of type \(\text{\texttt{\lin}}\) in the \(\text{\texttt{\fun\lin}}\) syntax, the function only accepts a linear value directly from the slot sequence.

\footnote{Although \(\text{\texttt{\lin}}\) and \(\text{\texttt{\bind}}\) should be abstract types, they could not be hidden because the code generated by the syntax extension will use them.}
let rec _traverse p =
  match p with
  | l is a lens pattern #l ->
      let rec _traverse p =
      | l is a variable pattern var
      | -> Data var
  | l is a constructor pattern C without parameters
      -> C
  | l is a constructor pattern with parameters
      -> C(l, l1, ..., l...) (with parameters)
  | _ -> throw an error

let conv (p : pattern) : pattern =
  if p is a lens pattern then
    _traverse p
  else
    Lin_. (_traverse p)

(*wrap it in the linear type constructor*)

Fig. 11 A translation for Lin-patterns.

val get_lin : ('a linlist, empty, 'pre, 'post) lens ->
  ('pre, 'post, 'a linlist) monad
let get_lin l = fun p ->
  l.put pre Empty, l.get pre

val put_lin : ('empty, 'a linlist, mid, 'post) lens ->
  ('pre, mid, 'a linlist) monad
let put_lin l m = fun pre ->
  match m with
  | mid, a -> l.put mid a, Data ()

val put_linval : ('empty, 'a linlist, 'pre, 'post) lens ->
  'a -> ('pre, 'post, unit data) monad
let put_linval l a = fun pre ->
  l.put pre a, Data ()

Fig. 13 APIs for direct slot access.

Here, iter' can accept the same polymorphic lens in its two arguments, like iter f _0 . _0.

4.3 Linearly Typed Lists

As an example of effective usage for pattern matching against linearly typed structured data, we introduce a linearly typed list type in Fig. 12. Furthermore, through this example, we introduce the functions get_lin, put_lin, and put_linval, which directly manipulate the slot pointed to by the lens, and the linear value constructor syntax to construct linear data.

Example 4.4 (Iterating over linearly typed lists) The following function (iter0 f) applies f to consume all elements in the list assigned to slot _0. Here, get_lin l takes a linear value from the slot referred to by l and then empties the slot.

val iter0 : ('a -> 'b) ->
  ('a linlist, empty, 'pre, 'post, 'x, unit data) monad
let rec iter0 f =
  match lin get lin _0 with
  | Cons(x, _0) -> f x, iter0 f
  | Nil -> return ()

Although iter0 is simple, it is not really flexible because the slot is fixed to _0 . However, taking a lens parameter like iter f l does not work.

Example 4.5 (A type error due to a monomorphic parameter) The following function (iter_fail) is not typeable:

let rec iter_fail f l =
  match lin get lin l with
  | Cons(x, _#l) -> f x; iter_fail f l
  | Nil -> return ()

This is because get_lin fixes the type of lens to ('a linlist, empty, 'pre, 'post) lens and cannot be used with type ('empty, 'a linlist, 'post, 'pre) lens, as required for a lens pattern allocating to the empty slot. Passing multiple lenses with different usages will work as follows:

val iter' : ('a -> 'b) ->
linearly typed list assigned to slot _0, building a new list at _1 by applying \( f \) in the reverse order.

```ocaml
val rev_map : ('a -> 'b) -> ('a linlist * all_empty, empty * ('b linlist * all_empty), unit data) monad
let rec loop () =
  match xlin get_lin _0 with
  | Cons(x, #_0) ->
    put_lin _1
    [xlinnet Cons(Data(f x), !!_1)] >>
    loop ()
  | Nil -> return ()
in
put_linval _1 Nil >>
loop ()
```

**Example 4.8 (Generalising map (1))** The following function `map'` generalises `map` in Example 4.6 to take a lens parameter that refers to the list it operates on.

```ocaml
val map' : ('a -> 'a) ->
  ('a linlist, empty, ('pre, 'mid) lens) ->
  (empty, ('a linlist, 'mid, 'pre) lens) ->
  ('pre, 'post, unit data) monad
let rec map' f s1 s2 =
  match xlin get_lin s1 with
  | Cons(x, #s2) ->
    map' f s1 s2 >>
    put_lin s2
    [xlinnet Cons(Data(f x), (!! s1))] |
  | Nil -> return ()
in
put_linval _1 Nil >>
loop ()
```

Unfortunately, `map'` cannot change the type of elements in the list. This is because, although we use two lenses `s1` and `s2`, one for extracting the source list from a slot and the other for assigning the destination list to another slot, they are shared for both the source type and the destination type.

**Example 4.9 (Generalising map (2))** By supplying different lenses for the source list and the destination list, we can define a generalised map that can take different types for the source and the destination.

```ocaml
val map : ('a -> 'b) ->
  ('a linlist, empty, ('pre, 'mid) lens) ->
  (empty, ('a linlist, 'mid, 'pre) lens) ->
  ('pre, 'post, unit data) monad
let rec map f s1 s2 s3 s4 =
  match xlin get_lin s1 with
  | Cons(x, #s2) ->
    map f s1 s2 s3 s4 >>
    put_lin s4
    [xlinnet Cons(Data (f x), (!! s3))] |
  | Nil -> return ()
```

5. An Encoding of Session Types

For a more practical example of linearly typed programming, we introduce session types and show a solution to the *Santa Claus problem* by utilising pattern matching on linearly typed structured values.

5.1 Session Types

Session types [8] can represent the communication protocol realised by a program and statically guarantee that communication proceeds and terminates safely. As with linear types, session types require linearity to track the number of times a session is used.

**Example 5.1 (An addition server)** Session-typed communication starts by establishing a session on a channel. The following program is a server that calculates the sum of two integers.

```ocaml
val ex5 : unit ->
  (((((close, int) send, int) recv, int) recv
lin
all_empty, unit) monad
let ex5 () =
  let xlin #_0, x = receive _0 in
  let ylin #_0, y = receive _0 in
  let lin #_0 = send _0 (x+y) in
  close _0
```

This program operates on the session assigned to slot _0, receiving two integers and sending their sum before terminating. Session types reflect such communication structure in types. Type \((\theta, \tau)\) `recv` denotes receiving a value of type \(\tau\) before behaving as session \(\theta\), whereas \((\theta, \tau)\) `send` denotes sending a value of type \(\tau\) before behaving as session \(\theta\), and `close` denotes the end of a session. This addition service has the following type at slot _0:\footnote{This type represents communication steps in right-to-left order owing to the postfix syntax of OCaml types.}

\[ (((close, int) send, int) recv, int) recv \]

**Figure 14** shows the signature of a communication API based on session types. The type \((\theta_1, \theta_2)\) channel is the type of a channel used as entry point of a session. A communication peer can wait for a peer with accept, and a session is established when a peer makes a connection request by request. Type \(\theta_1\) is the session that the accepting side (i.e., the server) must follow, whereas \(\theta_2\) is the one that the requesting side (client) must follow. In session type theory, if \(\theta_1\) and \(\theta_2\) are dual, one can ensure that the communication on that channel will be consistent (i.e., no deadlock may occur and the types of messages at each peer coincide). The duality on a channel can be established by constructing it using the API in Fig. 15. For example, `s2 finish` of type \(((close, \tau’) send \* (close, \tau’) recv) channel\) generates a channel that initially sends a value from the server to the client and then terminates.

```ocaml
val accept : ('s, 'c) channel ->
  ('pre, 'pre, 's lin) monad
val request : ('s, 'c) channel ->
  ('pre, 'pre, 'c lin) monad
```

\[ (((close, \tau’) send \* (close, \tau’) recv) channel\]
5.2 A Solution to the Santa Claus Problem

The Santa Claus problem [2], [27] is a problem in concurrent programming proposed by Tromo, and it has served as a benchmark for concurrent features in various programming languages [18]. We quote the problem from Ref. [27]:

Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeers being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. [...] the elves visit Santa in a group of three. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. [...] 

Modelling. We model the Santa Claus problem as follows. Santa Claus is waiting in the main thread by using accept on a channel available to the reindeers and elves. We assign each reindeer and elf a thread and let them establish a session with Santa by using request at a random rate. Santa will continue to communicate with all reindeers and finish the sessions when the number of established sessions with reindeers reaches nine. Similarly, Santa will finish the session with the elves when the number of their established sessions reaches three.

List of sessions. The sessions with reindeers and elves are stored in two linearly typed lists held by Santa. As a result, we can dynamically increase or decrease the number of sessions without losing linearity. Santa processes all sessions in a list at once using the function iter.

Figure 16 shows the linearly typed list in this example. The type \( \text{a slist} \) represents a list of sessions. Although similar to the linearly typed list in Fig. 12, type \( \text{a slist} \) differs from it in that it holds linearly typed content of type \( \text{a lin} \). Function iter is only used in the form iter \( i \ll f \). It is a function that passes the first \( i \) number of elements in the list referred to by \( \text{lens} \) to function \( f \). As we have seen in Examples 4.5, 4.8 and 4.9, it needs three parameters to refer to the same slot owing to the lack of polymorphism.

Figure 17 shows the Santa part of the solution to the problem (the elf and reindeer parts are relatively easy). Type kind is used to identify whether a peer is an elf or a reindeer. Lens _0 is used to temporarily store the session with reindeers or elves, and it is also used as a 'working slot' for iter. Lenses _1 and _2 store the lists for elves and reindeers, and we give them the aliases e and r, respectively.

First, Santa assigns empty lists to both e and r, and enters the loop. In the loop, Santa waits for a session by accept and receives the kind of the peer. If the peer is an elf, he stores the rest of the session in e and increases the count of the number of elves ecountr. Similarly, if the peer is a reindeer, he stores the session in r and increases rcountr. When either count reaches its limit, Santa sends the string "Let's deliver!" in close _0 and loop ch (ecountr) >> loop ch (rcountr) >> else loop ch (ecountr) >> loop ch (rcountr) >>

6. Related Works

A categorical framework for linear types in a parameterised monad was introduced by Atkey [1]. To the authors' knowl-
edge. Safeio, proposed in the post to the OCaml mailing list by Garrigue [6], is the first encoding of linearly typed resources by a parameterised monad in OCaml, and it is mostly reproduced in Section 3 of this paper.

6.1 Linear Types in Haskell Embedding based on De Bruijn indices. Similar in spirit to linocaml, Kiselyov’s finally tagless interpreters [13] are a technique for embedding a typed language into Haskell, and he showed an embedding of the typed lambda calculus and linear lambda calculus. For example, the function \( \lambda x. y \cdot x + y \), which calculates the sum of two linearly typed integers, can be written as \( \lambda m (\lambda (s z) z) \) by his technique. However, since de Bruijn indices use different numbers to represent the same variable, it is difficult for humans to grasp the binding structure of the program. Kiselyov also uses a parameterised monad as a basic technique for static typing, and his framework allows \( \lambda \)-abstraction, with the typing context growing when lambda abstractions are nested. He uses Haskell type classes to encode lambda abstractions.

Embedding based on HOAS. Polakow [24] encoded linear lambda calculus using Haskell’s type classes and functional dependencies. Since his technique offers a direct embedding based on HOAS [20], it does not need slot numberings as in this paper, and it avoids the readability problems of de Bruijn indices. The technique resembles Kiselyov’s; however, it differs from it in the representation of pre- and post-conditions, having only flags indicating whether a variable is used. Paykin and Zdancewic [19] extended Polakow’s method to Benton’s calculus [3] based on a linear/non-linear classification, offering a more flexible framework, in which they demonstrated many examples.

Since the existing works in Haskell depend on type classes and functional dependencies to encode variable usage in types, it is difficult to migrate them to other languages. Furthermore, they lack pattern matching against structured data.

6.2 Linearity in Session Type Implementations The first encoding of session types is attributed to Neubauer and Thiemann [17]. It is older than the above mentioned encodings, but also relies on Haskell’s functional dependencies. It only allows to handle one channel at a time and is difficult to generalise to multiple channels. Puccella and Tov [25] proposed a library implementation of session types that can handle multiple channels based on a parameterised monad. In their monad, pre- and post-conditions in the monad are a stack of linear resources, and the communication primitives apply on the top element of the stack. It also offers stack manipulation primitives dig and swap. This technique is applicable to languages other than Haskell. However, programming with such stack manipulations becomes cumbersome and tends to be unreadable. This problem was solved by Imai et al. [11] in Haskell using HOAS. Similar to Polakow’s technique, HOAS-based encoding can directly mention linear resources by variable name, thus making programs more readable. However, it is also difficult to adapt it to languages other than Haskell since it again requires type classes and functional dependencies.

6.3 Expressiveness Instead of lambda abstraction as in the linear lambda calculus, linocaml can pass linear values through slots pointed by lenses, which can be bound to variables in the host language.

An interesting question would be whether our library, which does not have linear \( \lambda \)-abstraction, has equal expressiveness to the ones by Kiselyov and Polakow.

First, let us consider the case where the linear argument is a non-functional, first-order value. linocaml can express the equivalent of linear abstractions and function applications such as \((\lambda x. y \cdot x + y) 42 21\) by defining add in the following way:

\[
\]

and by passing parameters via lenses as follows:

\[
\begin{align*}
\text{val} \text{linret 42} & \gg \text{fun} \text{lin} \_0 & \to \_1 \\
\text{val} \text{linret 21} & \gg \text{fun} \text{lin} \_1 & \to \_0
\end{align*}
\]

However, we have not considered how to introduce higher order functions such as \( \Lambda f. \lambda x. f x \). Since \text{fun} \text{lin} cannot be nested like \text{fun} \text{lin} \#_1 -> \text{fun} \text{lin} \#_2 -> , and because linocaml stores linear values in slots rather than in variables, it is not obvious how to encode such curried functions.

On the other hand, by using the host language abstraction mechanism, we can construct higher order functions by slot manipulation and lens passing, like iteriM, mapiM, as we have seen in Section 3.2.2 (Example 3.3 and Example 3.4). Furthermore, in Example 3.3, the function passed to iteriM updates the linear values by passing them via slot _1:

\[
\text{iteriM} (\text{fun} \ i x \to \text{lookup} i \_0 \_1 \gg \text{fun} \ y \to \text{update} i (x + y) \_0 \_1)
\]

In the linear lambda calculus, a closure that contains linear values must also be treated linearly. On the other hand, closures (\text{fun} \ i x \to ..) and (\text{fun} \text{lin} \#_0 \to ..) in linocaml do not have this limitation and can be used freely. Thus, although comparison of the expressiveness is not evident, the authors expect linocaml to be as expressive as the linear lambda calculus.

7. Conclusion This paper described an encoding of linear types in OCaml using a parameterised monad and lenses, which we have made available through the linocaml library. The usage of lenses as a handle to linear values allows easy porting to other languages such as Standard ML and Haskell. Additionally, we utilised OCaml’s syntax extension to provide pattern matching against linear values, which can be used to manipulate structured data such as linearly typed lists. For a practical example, we have shown a solution to the Santa Claus problem, which exploits linear pattern matching.

Notwithstanding its light weight, this encoding can simulate

\[\text{iteriM} \ (\text{fun} \ i x \to \text{lookup} i \_0 \_1 \gg \text{fun} \ y \to \text{update} i (x + y) \_0 \_1)\]

442

© 2019 Information Processing Society of Japan
static, linearly typed programming using a set of well-known features in functional programming such as monads and lenses. Linear types are still terra incognita for most programming languages with Rust being the only widely known programming language supporting them natively. By introducing them in OCaml, we enable programmers to directly benefit from resource safety, and, eventually, we hope that it will also bring runtime efficiency.

Future work. Since computation in LinMonad involves many closures, it is bound to be less efficient than programs written in direct style. Slot-based access also has a small cost to follow the nesting pairs. This cost would be negligible in communication-centric programs where the bottleneck lies in other parts; however, it does matter for computation-intensive tasks such as array manipulations. Such performance analysis and improvement are future works.

Lenses have a polymorphic type such as (\(a \to b, a \to * xs, b + * xs\)) lens for lens \_\_\_\_. However, such first-class polymorphic values are not available in many programming languages. Implementing this lens-based programming framework in Java-like languages, to allow wider use of the proposed techniques, is an interesting challenge.

Acknowledgments

We thank the anonymous reviewers for the thorough review and constructive comments. This work is partially supported by Kakenhi 16K00995 and 17K12662 from JSPS, Japan, and by Grants-in-aid for Promotion of Regional Industry-University-Government Collaboration from Cabinet Office, Japan.

References

Jacques Garrigue received his M.S. degree from University Paris 7 and his D.S. degree from the University of Tokyo in 1995. He is an alumnus of École Normale Supérieure in Paris. He was a Research Associate at Kyoto University from 1995 to 2004, and is now a Professor at Nagoya University. His interests are in the theory of programming languages, particularly type systems and proof of programs. He is a member of IPSJ, JSSST and ACM.