Regular Paper

Lightweight Linearly-typed Programming with Lenses and Monads

Keigo Imai[1,a)] Jacques Garrigue[2,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 [*1]. 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

[1] Faculty of Engineering, Gifu University, Gifu 501–1193, Japan
[2] Graduate School of Mathematics, Nagoya University, Nagoya, Aichi 464–8602, Japan
[a)] keigoi@gifu-u.ac.jp
[b)] garrigue@math.nagoya-u.ac.jp

[*1] Polakow [24] and other authors also use a parameterised monad that encodes the pre- and post-conditions on linearity (See Section 6).

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/keigoi/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`.

```
val alloc : 'a list -> 'a larr
val dealloc : 'a larr -> unit
val lookup : int -> 'a larr -> 'a larr * 'a
val update : int -> 'a -> 'a larr -> 'a larr
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.

```
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 [*2].

□

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

```
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

[*2] 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, unit 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.

```
type ('pre, 'post, 'a) monad
type 'a lin = Lin__ of 'a
type 'a data = Data of 'a
val return : 'a -> ('pre, 'pre, 'a data) monad
val (>>=) : ('pre, 'mid, 'a data) monad ->
      ('a -> ('mid, 'post, 'b) monad) ->
      ('pre, 'post, 'b) monad
val (>>) : ('pre, 'mid, 'a data) monad ->
      ('mid, 'post, 'b) monad ->
      ('pre, 'post, 'b) monad
type empty = Empty
val run :
  (unit -> (empty, empty, 'a data) monad) ->
  'a
```

Fig. 2 A parameterised monad LinMonad.

crucial role in the pattern matching introduced in Section 4.

The function `alloc` creates an array. Its return type

```
(empty, 'a larr, unit data) monad
```

says that it consumes an empty value (of type `empty`) as input, allocates an array of type `'a larr`, outputs it, and returns the unit value of type unit as the result of the computation. On the other hand, `dealloc` deallocates an input array, and its output is an empty value. The operation `lookup` i extracts the i-th element of the input array of type `'a larr` and returns it with type `'a`, while outputting the unchanged array. Functions `map` and `to_list` also correspond to Example 2.1 in a similar way.

Figure 2 shows the type signature for `LinMonad`. The types `'a lin` and `'a data` wrap a value with the constructors `Lin__` and `Data`, respectively. `Lin__` may be utilised to implement linearly typed APIs; however, it must not be used by the end users [*3].

Function `return` is a "pure" command that does not change the input value and outputs it as it is to the next command; hence, the pre- and post-conditions have the same type `'pre`. For coherence with Section 4, the result type of `return` is wrapped with the type constructor **data**, which is removed in the following bind (>>=) and `run` operations.

The property that linear values are never discarded is guaranteed by the type of bind (and of `run`, which is shown later). The bind operation roughly corresponds to the pipe mechanism in UNIX shells. It applies the function on the right hand side (rhs) to the result value of the command on the left hand side (lhs) and, at the same time, passes the linear output value from the lhs to the command obtained from the rhs. The type signature requires the type constructor **data** to be removed from the type of the result `'a data` in the lhs and also the output type `'mid` of the lhs to be matched with the input type of the rhs. Thus, if the rhs requires its input to be `empty` while the lhs outputs a linear value of type `'a lin`, it is statically detected as a type error. As a whole, the composed monadic value takes an input of type `'pre` required by the lhs, outputs a value of type `'post` from the rhs, and returns the result of type `'b` from the rhs.

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 [*4]. 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 [*5].

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.

```
val ex1: unit -> (empty,empty,unit data) monad
let ex1 () =
  alloc [100; 200; 300] >>
  map string_of_int >>
  update 1 "Hello" >>
  lookup 1 >>= fun x ->
  dealloc >>
  (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 [*6]. □

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:

```
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.

[*3] However, within the framework described in Section 3, it is not harmful to use `Lin__` since there are no means to take linear values out of monadic values.

[*4] The operator precedence is as follows: e_1 >>= (**fun** x -> e_2).

[*5] The let-binding is required because OCaml is not pure, and the expression e_2 may have side effects.

[*6] 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 `(x₀,(x₁,(x₂, ...)))`. 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 `empty * (empty * (empty * ..))`. There are two ways in OCaml to write types with such an infinite structure.

(1) OCaml's polymorphic variants or objects allow equi-recursion on types [22]. By using a polymorphic variant constructor `` `cons``, we can write the infinite type

```
[`cons of empty * [`cons of empty * ..]]
```

as

```
[`cons of empty * 't] as 't
```

where T `as 't` is an equi-recursive type identical to T[(T `as 't`)/`'t`], where the type variable `'t` in T is replaced by T `as 't`. Object types can also encode an infinite sequence as `<cons : empty * 't> as 't`.

(2) The `-rectypes` option of the OCaml compiler enables to construct equi-recursive types through arbitrary type constructors so that `empty * (empty * (empty * ..))` can be written more directly as `empty * 't as '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 `lens` library [14] and Pickering's lenses [21] allow writing back values of different types.

Figure 3 shows the definition of lenses for slot manipulation.

- The lens is a pair of a view function `get` and a putback function `put`. The type parameters `'d1` and `'d2` represent the type of the source to be referenced by the lens. The function `get` returns the view of type `'a` from the source `'d1`. On the other hand, the function `put` functionally updates the source `'d1` to the type `'d2` by writing back a value of type `'b`.
- Lenses enable to refer to arbitrary finite positions in a slot sequence, with the linearity being enforced by the monad. Lens `_0` refers to the 0-th element [*7] of a slot sequence, i.e. it points to the lhs of a pair. Similarly, lenses `_1` and `_2` refer to the first and second elements in a slot sequence, respectively.
- The third and subsequent elements in a slot sequence are obtained by the function `succ`, which builds a new lens that refers to the next element of an existing lens. For example, lenses that refer to the first and second elements can also be written as `succ _0` and `succ (succ _0)`, respectively. However, because OCaml enforces the value restriction, the type of such expressions becomes monomorphic and cannot be used at multiple types. This can be avoided by using GADTs, as shown in Section 3.2.4.

```
type ('a, 'b, 'd1, 'd2) lens =
  {get: 'd1 -> 'a; put: 'd1 -> 'b -> 'd2}
val _0 : ('a, 'b, 'a * 'xs, 'b * 'xs) lens
let _0 =
  {get = (fun (a,_) -> a);
   put = (fun (_,xs) b -> (b,xs))}
val _1 : ('a, 'b, 'x1 * ('a * 'xs),
          'x1 * ('b * 'xs)) lens
let _1 =
  {get=(fun(_,(a,_)) -> a);
   put=(fun(x,(_,xs)) b -> x,(b,xs))}
val _2 : ('a, 'b, 'x1 * ('x2 * ('a * 'xs)),
          'x1 * ('x2 * ('b * 'xs))) lens
let _2 =
  {get=(fun(_,(_,(a,_))) -> a);
   put=(fun(x,(y,(_,xs))) b -> x,(y,(b,xs)))}
val succ : ('a, 'b, 'xs, 'ys) lens
  -> ('a, 'b, ('x * 'xs), ('x * 'ys)) lens
let succ l =
  {get = (fun (_,xs) -> l.get xs);
   put = (fun (x,xs) b -> (x, l.put xs b))}
```

Fig. 3 Lenses for manipulating slots.

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:

```
val all_emp : empty * 't as 't
let rec all_emp = Empty, all_emp
```

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

```
val slots1 : int larr * (empty * 't as 't)
let slots1 = _0.put all_emp arr
```

The composite lens `succ (succ _0)` in the following code assigns an array `arr1` of type `string larr` to the second position in the sequence.

```
val slots2 : int larr * (empty * (string larr*
                  (empty * 't as 't)))
let slots2 = (succ (succ _0)).put slots1 arr1
```

□

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 `@>` which updates an element in the slot sequence using a command. Expression `m @> l` executes the computation `m` in the slot referenced by lens `l`. The type signature reads as follows:

- From the pre-condition `'pre` of `m @> l`, a linear resource of type `'p` is obtained using lens `l`.
- The resource `'p` is consumed in the monadic computation `m`, and the post-condition `'q` is produced and the result value `'a` is returned.

```
type all_empty = empty * 't as 't
val run':
  (unit -> (all_empty, all_empty, 'a data) monad)
    -> 'a
val (@>) : ('p, 'q, 'a) monad
  -> ('p, 'q, 'pre, 'post) lens
  -> ('pre, 'post, 'a) monad
```

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

[*7] 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 `m @> l`.

The following example handles multiple linear resources using lenses. We introduce the functions `iteriM` and `mapiM` as variants of `List.iter` and `List.map` in OCaml, respectively. For instance, `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)
      _0 >>
  to_list @> _1 >>= fun xs ->
  dealloc @> _0 >>
  dealloc @> _1 >>
  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 `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 `fun i x -> ...` passed to `iteriM` reads the `i`-th element of the first array by calling `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 @> _1 >>
  dealloc @> _0 >>
  return xs
```

□

Figure 5 shows the type signatures of `iteriM` and `mapiM`, which represent the following linearity constraints:

- In `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 `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 `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 lens `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.

```
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, 'post, unit data) monad
```

Fig. 5 Signatures for `iteriM` and `mapM`.

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 `extend` and `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 `extend` and `shrink` enable to handle as many slots as required.

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

```
val ex4 : unit ->
  (empty,empty,string list data) monad
let ex4 () =
  extend >>
  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 `succ _0` 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 `succ (succ _0)`, which is cumbersome.

Lenses defined using generalised algebraic data types (GADTs) [7] constructors as in **Fig. 6** can avoid the value restriction, keeping such combinations polymorphic.

`Fst` is a lens that refers to the 0-th slot, and `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.

```
type (_,_,_,_) lens =
  | Fst  : ('a,'b,'a * 'xs, 'b * 'xs) lens
  | Next : ('a,'b,'xs,'ys) lens -> ('a,'b,'x * 'xs, 'x * 'ys) lens
  | Any  : ('d1 -> 'a) * ('d1 -> 'b -> 'd2) -> ('a, 'b, 'd1, 'd2) lens

val lget : ('a, 'b, 'xs, 'ys) lens -> 'xs -> 'a
let rec lget : type a b xs ys. (a, b, xs, ys) lens -> xs -> a = fun ln xs ->
  match ln,xs with
  | Fst, (a,_)        -> a
  | Next ln', (_,xs') -> lget ln' xs'
  | Any (get, _), xs  -> get xs

val lput : ('a, 'b, 'xs, 'ys) lens -> 'xs -> 'b -> 'ys
let rec lput : type a b xs ys. (a,b,xs,ys) lens -> xs -> b -> ys = fun ln xs b ->
  match ln, xs with
  | Fst, (_, xs) -> (b, xs)
  | Next ln', (a, xs') -> (a, lput ln' xs' b)
  | Any (_, put), xs -> put xs b

let _0 = Fst;;  let _1 = Next _0;; let _2 = Next _1;; let _3 = Next _2
```

Fig. 6 A GADT-based construction of lenses.

```
type ('pre, 'post, 'a) monad= 'pre -> 'post * 'a
let return a = fun pre -> pre, Data a
let m >>= f = fun pre ->
  match m pre with
  | mid, Data a -> f a mid
let m1 >> m2 = fun pre ->
  match m1 pre with
  | mid, Data _ -> m2 mid
let run f =
  match f () Empty with
  | Empty, Data a -> a
let run' f  =
  match f () all_emp with
  | _, Data a -> a
let (@>) m l = fun pre ->
  match m (l.get pre) with
  | q, d -> l.put pre q, d
```

Fig. 7 An implementation of `LinMonad`.

that refers to the next element of *l* which is equivalent to `succ l`. `Any (`*get*`, `*put*`)` is a lens consisting of an arbitrary view function *get* and putback function *put*. `_0, _1, _2, _3` are defined using constructors, and therefore, they are not subject to the value restriction and are polymorphic.

`lget` and `lput` are the view operation and putback operation, respectively. Here, `type a xs`, etc. in type annotations are locally abstract types and represent types to be refined by pattern matching against GADT constructors.

3.3 Implementations of the Monad and APIs

`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 `('pre, 'post, 'a) monad` denotes a state monad with state changing from `'pre` to `'post`, implemented as the function type `'pre -> 'post * 'a`.

Monadic operations `return`, `>>=`, `>>`, `run` are standard; however, they wrap the result value with a `Data` constructor. The function `run` explicitly handles the state value `Empty` to ensure that the pre- and post-conditions will be `empty`. Here, `run'` is almost the same except that it uses `all_emp` instead of `Empty`. The operator `@>` executes the monad value `m` in the environment obtained by applying the lens `l` to `pre`, and it updates the slot sequence with that lens.

Implementing a linearly typed API When implementing a

```
type 'a larr = Obj.t array lin
let alloc xs = fun Empty ->
  let arr =
    Array.make (List.length l) (Obj.repr 0) 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, Data (Obj.obj arr.(i))
let update i a = fun ((Lin__ arr) as pre) ->
  arr.(i) <- (Obj.repr a);
  pre, Data ()
let map f = fun (Lin__ arr) ->
  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, Data (Obj.magic (Array.to_list arr))
```

Fig. 8 An implementation of linearly typed arrays.

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.

```
val Obj.repr : 'a -> Obj.t
val Obj.obj : Obj.t -> 'a
val Obj.magic : 'a -> 'b
```

Here, type `Obj.t` is used to embed arbitrary types.

The internal representation of OCaml arrays is dynamically determined by the value passed to the initialisation function. To avoid creating a specialised `float` array, `alloc` initialises an array with a value of `0` of type `int` before copying the contents of the list. Looking up an array (`lookup` and `map`) restores the actual type from `Obj.t` type by using `Obj.obj`. Updating an array (`update` and `map`) stores the elements converted to `Obj.t` by using

`Obj.repr`. Function `to_list` converts a list of type `Obj.t list` to its true type by using `Obj.magic`.

Since each function in the API handles arrays linearly and does not add any reference (e.g., assignment to a global variable), this API guarantees linear access to all arrays.

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 [*9]. The expression **`let%lin`** *pat* = e_1 **`in`** e_2 executes e_1, binds its result to pattern *pat*, and then executes e_2. Pattern *pat* is extended to include *lens pattern* **#***l*, which assigns the matched linear value into an empty slot referred to by *l* [*10].

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 linearly typed results of type `('pre, 'post, 'a lin) monad`.

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 [*11].
- Functions other than `alloc` 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,` τ`) 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

```
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 l = 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 arr0) = l.get pre in
  l.put pre Empty, Lin__ (arr0, Data(arr.(i)))

val update : int -> 'a ->
  ('a larr, empty, 'pre, 'post) lens ->
  ('pre, 'post, 'a larr) monad
let update i a l = fun pre ->
  let ((Lin__ arr) as arr0) = l.get pre in
  arr.(i) <- a;
  l.put pre Empty, arr0

val map : ('a -> 'b) ->
  ('a larr, empty, 'pre, 'post) lens ->
  ('pre, 'post, 'b larr) monad
let map f l = fun pre ->
  let (Lin__ arr) = l.get pre in
  l.put pre Empty, Lin__ (Array.map f arr)
```

Fig. 9 A linearly typed array API for lens patterns.

array operation of Example 2.2 as follows:

```
val ex1' : unit ->
  (all_empty, all_empty, unit data) monad
let ex1' () =
  (* Variable arr referring to the 0-th slot *)
let arr = _0 in
let%lin #arr = alloc [100; 200; 300] in
let%lin #arr = map string_of_int arr in
let%lin #arr = update 1 "Hello" arr in
let%lin #arr, x = lookup 1 arr in
dealloc 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:

```
val ex4' : unit ->
  (all_empty,all_empty,string list data) monad
let ex4' () =
  let s = _0 and t = _1 in
  let%lin #s = alloc [100; 200; 300] in
  let%lin #t = alloc ["abc"; "def"; "ghi"] in
  let%lin #s =
    mapiM (fun i x ->
        let%lin #t, str = lookup i t in
        return (str ^ string_of_int x)) s in
  let%lin #s, xs = to_list s in
  dealloc t >>
  dealloc s >>
  return xs
```

□

4.2 A Semantics for Lens Patterns

We give the semantics of the lens pattern and **`%lin`** by macro

[*9] Here, **`%lin`** denotes a syntax extension point in OCaml and the preprocessor expands it into a vanilla OCaml syntax tree without the extension.

[*10] We replace the pattern **`#t`**, which is originally the syntax for matching against all values of the polymorphic variant type `t`.

[*11] Note that `'a larr` is an alias for `'a array lin` (see Fig. 1).

expansion. First, let us consider the case where `%lin` contains a lens pattern only and generalise it for structured patterns.

Expressions with lens patterns

let%lin #l = e_1 **in** e_2

are treated as an abbreviation of the following expression:

e_1 >>- **fun%lin** #l -> e_2

Here, >>- is a variant of the bind function, which requires a **fun%lin** function on its right-hand side.

The function **fun%lin** #l -> e_2 is expanded to

```
Bind__ (fun tmp -> _put l tmp >> e2)
```

by the preprocessor, where *tmp* is a fresh variable. `Bind__` is a constructor that distinguishes a **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 `%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 `%lin` has the following form:

```
(fun%lin #l -> e2) :
  (α lin -> (pre, post, β) monad) bind
```

where α, *pre*, *post*, and β are determined by the type of lens l and by expression e_2. For example, an expression with lens `_0` would have the following type:

```
(fun%lin #_0 -> return ()) :
  ('a lin ->
    (empty * 'pre, 'a lin * 'pre, unit data)
      monad) 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 **let%lin** #l = e_1 **in** e_2 is expanded to e_1 >>- `Bind__(`**fun** *tmp* -> `_put` l *tmp* >> 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 `%lin`. Operator >>- is a specialised bind function that takes a function wrapped by `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 `lookup`. Special care is needed for variable patterns as there is a risk that a linear value of type α **lin** might be bound to a variable pattern that has a polymorphic type. Here, we assume that the top level of the pattern of **fun%lin** is a linear value and that the unlimited value appearing within it is wrapped

```
type 'f bind = Bind__ of 'f

val (>>-) : ('pre, 'mid, 'a lin) monad
  -> ('a lin -> ('mid, 'post, 'b) monad) bind
  -> ('pre, 'post, 'b) monad
let (>>-) m (Bind__ f) = fun pre ->
  match m pre with
  | mid, a -> f a mid

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

Fig. 10 Functions for the `%lin` syntax extension.

with `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 `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 `Data`, we use the following rules for pattern expansion:

- Variable patterns are allowed inside the `Data` constructor, like `Data x`.
- Variable patterns outside `Data` are implicitly wrapped by `Data` to have type τ **data**.

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

We further introduce syntax extensions **match%lin** and **function%lin**, which extend **match** and **function** with lens patterns, respectively. We summarise the expansion of each syntax extension as follows:

- **fun%lin** *pat* -> e expands to
  `Bind__ (`**fun** *conv(pat)* -> *puts(pat)* >> e).
  Here, *conv(pat)* is a syntactic function for converting the pattern *pat*, which is given later. *puts(pat)* is a function that generates a command
  `_put` l_1 tmp_1 >> $\cdots$ >> `_put` l_n tmp_n
  for the lens $l_1, \ldots, l_n$ appearing in *pat*, where the variables $tmp_1, \ldots, tmp_n$ are freshly generated by *conv(pat)*.
- **function%lin** $case_1$ | $\cdots$ | $case_n$ expands each clause $case_i$ of the form pat_i -> e_i as in **fun%lin**.
- **let%lin** *pat* = e_1 **in** e_2 is expanded similarly to the case of e_1 >>- **fun%lin** *pat* e_2.
- **match%lin** e **with** $case_1$ | $\cdots$ | $case_n$ is expanded similarly to e >>- **function%lin** $case_1$ | $\cdots$ | $case_n$.

Figure 11 shows a pseudo-OCaml code for expanding `%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 `Data x` of type `'a` **data**. For a constructor pattern, it converts the argument pattern recursively. An error occurs for patterns leaking linear values, e.g,. **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 **lin** are used linearly, provided she/he does not directly use the constructors `Lin__` and `Bind__`[*12]. 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 **lin** in the **fun%lin** syntax, the function only accepts a linear value directly from the slot se-

[*12] Although **lin** and **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
  | is a lens pattern #l
    -> Generate a fresh variable tmp;
       Record a pair of lens l and variable tmp;
       tmp
  | is a variable pattern var
    -> Data var
    | is a constructor pattern C without parameters
    -> C
  | is a constructor pattern with parameters
    C(p1,p2,...)
    -> C(_traverse p1, _traverse p2, ...)
  | _
    -> report 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.

```
type 'a linlist = 'a linlist_ lin
 and 'a linlist_ =
     Cons of 'a data * 'a linlist | Nil
```

Fig. 12 A linearly typed list.

quence via the monad and its linear components are immediately put back in the slot sequence.

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 * 'xs, empty * 'xs, 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) ->
  ('a linlist, empty, 'pre, 'post) lens ->
  (empty, 'a linlist, 'post, 'pre) lens ->
  ('pre, 'post, unit data) monad
let rec iter' f l1 l2 =
  match%lin get_lin l1 with
  | Cons(x, #l2) -> f x; iter' f l1 l2
  | Nil -> return ()
```

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

Linear value constructor. To define a `map` function on linearly typed lists, we need a means to construct a linear value. We introduce the syntax `[%linret` *c*`]` for this:

- `[%linret` *c*`]` is a monadic value with result value *c*.
- In `[%linret` *e*`]`, only nested application of constructors `C(`c_1`, ..., `c_n`)` or *lens references* are allowed in *e*.
- The lens reference `!!` *l* returns the value of the non-empty slot referred to by lens *l*.
- `[%linret` *e*`]` empties the slots referred to by the lens references `!!` $l_1, \ldots,$ `!!` l_n occurring in *e*.

Using these, we can define a `map` function on lists.

Example 4.6 (`map` on linearly typed lists) The function `map0 f` consumes the linearly typed list assigned to `_0` and produces a new list in `_0` that is obtained by applying `f` to each element in the given list.

```
val map0 : ('a -> 'b) ->
  ('a linlist * 'xs, 'b linlist * 'xs, unit data)
    monad
let rec map0 f =
  match%lin get_lin _0 with
  | Cons(x, #_0) ->
    map0 f >>
    put_lin _0 [%linret Cons(Data(f x), !!_0)]
  | Nil -> put_linval _0 Nil
```

Here, `put_lin` *l m* executes *m* and assigns the resulting value to the slot referred to by lens *l*. Expression `put_linval` *l v* assigns the value *v* to the slot referred to by *l*. □

Directly handling values in slots **Figure 13** shows the signatures and implementations of the functions `get_lin`, `put_lin`, and `put_linval`.

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

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

val put_linval :
  (empty,'a lin,'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.

Example 4.7 (Tail-recursive `map`) `List.rev_map` in the OCaml standard library is a tail-recursive variant of `map` in which the call stack does not grow linearly. The following function (`rev_map f`) is analogous to `List.rev_map` and operates on the linearly typed list assigned to slot `_0`, building a new list at `_1` by applying `f` in the reverse order.

```
val rev_map : ('a -> 'b) ->
  ('a linlist * all_empty,
   empty * ('b linlist * all_empty),
   unit data) monad
let rev_map f =
  let rec loop () =
    match%lin get_lin _0 with
    | Cons(x, #_0) ->
       put_lin _1
         [%linret 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 `map0` in Example 4.6 to take a lens parameter that refers to the list it operates on.

```
val map' : ('a -> 'a) ->
  ('a linlist, 'pre, 'mid) lens ->
  (empty, 'a linlist, 'mid, 'pre) lens ->
  ('pre, 'pre, unit data) monad
let rec map' f s1 s2 =
  match%lin get_lin s1 with
  | Cons(x, #s2) ->
     map' f s1 s2 >>
     put_lin s2
       [%linret Cons(Data(f x), !! s1)]
  | Nil -> put_linval s2 Nil
```

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.

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

□

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.

```
val ex5 : unit ->
  (((((close , int) send, int) recv, int) recv
    lin * all_empty,
   all_empty,
   unit session) monad
let ex5 () =
  let%lin #_0, x = receive _0 in
  let%lin #_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 τ before behaving as session θ, $(\theta,\ \tau)$ `send` denotes sending a value of type τ before behaving as session θ, and `close` denotes the end of a session. This addition service has the following type at slot `_0`: [*13]

```
(((close, int) send, int) recv, int) recv lin
```

□

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 θ_1 is the session that the `accept`ing side (i.e., the server) must follow, whereas θ_2 is the one that the `request`ing side (client) must follow. In session type theory, if θ_1 and θ_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, `s2c finish` of type `((close, 'v) send * (close, 'v) recv) channel` generates a channel that initially sends a value from the server to the client and then terminates.

```
type ('s, 'v) send    type ('s, 'v) recv
type close
type ('s, 'c) channel

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

val send :
  (('s, 'v) send lin, empty, 'pre, 'post) lens
  -> 'v -> ('pre, 'post, 's lin) monad
val receive :
  (('s, 'v) recv lin, empty, 'pre, 'post) lens
  -> ('pre, 'post, ('s lin * 'v data) lin) monad
val close :
  (close lin, empty, 'pre, 'post) lens
  -> ('pre, 'post, unit data) monad
```

Fig. 14 Communication API with session types.

*13 This type represents communication steps in right-to-left order owing to the postfix syntax of OCaml types.

```
val s2c : ('s * 'c) channel ->
    (('s, 'v) send * ('c, 'v) recv) channel
val c2s : ('s * 'c) channel ->
    (('s, 'v) recv * ('c, 'v) send) channel
val finish : (close * close) channel
```

Fig. 15 The channel creation API for ensuring duality.

```
type 'a slist_ = SCons of 'a lin * 'a slist | SNil
 and 'a slist = 'a slist_ lin

val iter : int ->
 ('a slist, empty, 'pre, empty*'mid0) lens ->
 (empty, 'a slist, 'a lin*'mid0, 'mid) lens ->
 (empty, 'a slist, empty*'mid0, 'pre) lens ->
 (unit -> ('mid, 'pre, unit data) monad) ->
 ('pre, 'pre, unit data) monad
let rec iter i l1 l2 l3 f =
  if i=0 then
    return ()
  else
    match%lin get_lin l1 with
    | SCons(#_0, #l2) ->
       f () >>
       iter (i-1) l1 l2 l3 f
    | SNil ->
       put_linval l3 SNil
```

Fig. 16 Iteration on a list of sessions.

5.2 A Solution to the Santa Claus Problem

The Santa Claus problem [2], [27] is a problem in concurrent programming proposed by Trono, 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 θ `slist` represents a list of sessions. Although similar to the linearly typed list in Fig. 12, type `slist` differs from it in that it holds linearly typed content of type θ `lin`. Function `iter` is only used in the form `iter i l l l f`. It is a function that passes the first `i` number of elements in the list referred to by lens `l` to function `f`. As we have seen in Exampless 4.5, 4.8 and 4.9, it needs three parameters to refer to the same slot owing to the lack of polymorphism.

```
type kind = Elf | Reindeer
type santa_ch =
  (((close, string) send, kind) recv *
   ((close, string) recv, kind) send)
  channel

let e = _1 and r = _2

val loop : santa_ch -> (int * int) ->
  (empty * ((string, close) send slist *
            ((string, close) send slist * _)),
   _, _) monad
let rec loop ch (ecount,rcount) =
  let%lin #_0 = accept ch in
  (match%lin receive _0 with
  | #_0, Elf ->
     put_lin e [%linret SCons(!!_0, !!e)] >>
     return (ecount+1, rcount)
  | #_0, Reindeer ->
     put_lin r [%linret SCons(!!_0, !!r)] >>
     return (ecount, rcount+1))
  >>= fun (ecount, rcount) ->
  if rcount=9 then
    iter 9 r r r (fun () ->
      let%lin #_0 = send _0 "Let's deliver!"
      in close _0) >>
    loop ch (ecount,0)
  else if ecount=3 then
    iter 3 e e e (fun () ->
      let%lin #_0 = send _0 "Make a new toy!"
      in close _0)  >>
    loop ch (ecount-3,rcount)
  else
    loop ch (ecount,rcount)

val santa : santa_ch -> (all_empty,_,_) monad
let santa ch =
  put_linval e SNil >>
  put_linval r SNil >>
  loop ch (0,0)
```

Fig. 17 A solution for the Santa Claus problem using lists of sessions.

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 `receive`s 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 `ecount`. Similarly, if the peer is a reindeer, he stores the session in `r` and increases `rcount`. When either count reaches its limit, Santa sends the string `"Let's deliver!"` if there are nine reindeers or the string `"Make a new toy!"` if there are three or more elves, and then terminates the sessions.

6. Related Works

A categorical framework for linear types in a parameterised monad was introduced by Atkey [1]. To the authors' knowledge, `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 $\lambda x.\lambda y.x + y$, which calculates the sum of two linearly typed integers, can be written as `lam (lam (add (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 λ 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. Pucella 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 λ-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 $(\lambda x.\lambda y.x + y)$ 42 21 by defining `add` in the following way:

```
val add : (int lin, empty, 'pre, 'mid) lens ->
      (int lin, empty, 'mid, 'post) lens ->
      ('pre, 'post, int lin) monad
```

and by passing parameters via lenses as follows:

```
[%linret 42] >>- fun%lin _0 ->
[%linret 21] >>- fun%lin _1 ->
add _0 _1
```

However, we have not considered how to introduce higher order functions such as $\lambda f.\lambda x.f\,x$. Since **`fun%lin`** cannot be nested like in **`fun%lin`** $\#l_1$ `->` **`fun%lin`** $\#l_2$ `->` , and because `linocaml` stores linear values in slots rather than in variables, it is not obvious how to encode such curried functions [*14].

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`:

```
iteriM (fun i x ->
    lookup i @> _1 >>= fun y ->
    update i (x + y) @> _1)
    _0
```

In the linear lambda calculus, a closure that contains linear values must also be treated linearly. On the other hand, closures `(fun i x -> ..)` and `(fun%lin #_0 -> ..)` 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

[*14] For example, by using `extend` and `shrink` in Section 3.2.3, we can construct a local environment at the beginning of the slot sequence. However, it leads us to de Bruijn indices, with the same readability problems as in Kiselyov's encoding.

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, 'b, 'a * 'xs, 'b * 'xs) lens` for lens `_0`. 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 16K00095 and 17K12662 from JSPS, Japan, and by Grants-in-aid for Promotion of Regional Industry-University-Government Collaboration from Cabinet Office, Japan.

References

[1] Atkey, R.: Parameterized Notions of Computation, *Journal of Functional Programming*, Vol.19, No.3-4, pp.335–376 (online), DOI: 10.1017/S095679680900728X (2009).

[2] Ben-Ari, M.: How to solve the Santa Claus problem, *Concurrency: Practice & Experience*, Vol.10, No.6, pp.485–496 (online), DOI: 10.1002/(SICI)1096-9128(199805)10:6<485::AID-CPE329>3.0.CO;2-2 (1998).

[3] Benton, P.N.: A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models, *Computer Science Logic, 8th International Workshop*, LNCS, Vol.933, pp.121–135, Springer (online), DOI: 10.1007/BFb0022251 (1994).

[4] Bernardy, J.-P., Boespflug, M., Newton, R.R., Jones, S.P. and Spiwack, A.: Linear Haskell: Practical linearity in a higher-order polymorphic language, *PACMPL*, Vol.2, No.POPL, pp.5:1–5:29 (online), DOI: 10.1145/3158093 (2018).

[5] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C. and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM Trans. Program. Lang. Syst.*, Vol.29, No.3, p.17 (online), DOI: 10.1145/1232420.1232424 (2007).

[6] Garrigue, J.: Safeio (A mailing-list post) (2006), available from ⟨https://github.com/garrigue/safeio⟩.

[7] Garrigue, J. and Normand, J.L.: Adding GADTs to OCaml: The direct approach, *ACM SIGPLAN Workshop on ML* (2011), available from ⟨https://www.math.nagoya-u.ac.jp/~garrigue/papers/ml2011.pdf⟩.

[8] Honda, K., Vasconcelos, V.T. and Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming, *ESOP '98: Proc. 7th European Symposium on Programming*, LNCS, Vol.1381, pp.122–138, Springer (online), DOI: 10.1007/BFb0053567 (1998).

[9] Imai, K., Yoshida, N. and Yuen, S.: Session-ocaml: A Session-Based Library with Polarities and Lenses, *COORDINATION 2017: Coordination Models and Languages*, LNCS, Vol.10319, pp.99–118, Springer (online), DOI: 10.1007/978-3-319-59746-1_6 (2017).

[10] Imai, K., Yoshida, N. and Yuen, S.: Session-ocaml: A Session-based Library with Polarities and Lenses, *Sci. Comput. Program.*, Vol.172, pp.135–159 (online), DOI: 10.1016/j.scico.2018.08.005 (2018). To appear.

[11] Imai, K., Yuen, S. and Agusa, K.: Session Type Inference in Haskell, *PLACES 2010: 3rd Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, Vol.69, pp.74–91, EPTCS (online), DOI: 10.4204/EPTCS.69.6 (2010).

[12] Jones, M.P.: Type Classes with Functional Dependencies, *ESOP '00: Proc. 9th European Symposium on Programming Languages and Systems*, LNCS, Vol.1782, pp.230–244, Springer (online), DOI: 10.1007/3-540-46425-5_15 (2000).

[13] Kiselyov, O.: Typed Tagless Final Interpreters, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22–26, 2010, Revised Lectures*, LNCS, Vol.7470, pp.130–174, Springer (online), DOI: 10.1007/978-3-642-32202-0_3 (2010).

[14] Kmett, E.: Lenses, Folds and Traversals (2012), available from ⟨http://lens.github.io/⟩.

[15] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. and Vouillon, J.: Representation of OCaml data types, *The OCaml System Release 4.07 Documentation and User's Manual* (2018), available from ⟨http://caml.inria.fr/pub/docs/manual-ocaml-4.07/intfc.html⟩.

[16] Marlow, S.: Haskell 2010 Language Report (2010), available from ⟨https://www.haskell.org/definition/⟩.

[17] Neubauer, M. and Thiemann, P.: An Implementation of Session Types, *PADL '04: Practical Aspects of Declarative Languages*, LNCS, Vol.3057, pp.56–70, Springer (online), DOI: 10.1007/978-3-540-24836-1_5 (2004).

[18] Benton, N.: Jingle Bells: Solving the Santa Claus Problem in Polyphonic C♯ (2003), available from ⟨https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/santa.pdf⟩.

[19] Paykin, J. and Zdancewic, S.: The linearity Monad, *Proc. 10th ACM SIGPLAN International Symposium on Haskell*, pp.117–132, ACM (online), DOI: 10.1145/3122955.3122965 (2017).

[20] Pfenning, F. and Elliot, C.: Higher-Order Abstract Syntax, *PLDI '88: Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp.199–208, ACM (online), DOI: 10.1145/53990.54010 (1988).

[21] Pickering, M., Gibbons, J. and Wu, N.: Profunctor Optics: Modular Data Accessors, *The Art, Science, and Engineering of Programming*, Vol.1, No.2, p. Article 7 (online), DOI: 10.22152/programming-journal.org/2017/1/7 (2017).

[22] Pierce, B.C.: Recursive Types, *Types and Programming Languages*, MIT Press, chapter 20 (2002).

[23] Plasmeijer, R., van Eekelen, M. and van Groningen, J.: Clean Version 2.2 Language Report (2011), available from ⟨https://clean.cs.ru.nl/Clean⟩.

[24] Polakow, J.: Embedding a Full Linear Lambda Calculus in Haskell, *Haskell '15: Proc. 2015 ACM SIGPLAN Symposium on Haskell*, pp.177–188, ACM (online), DOI: 10.1145/2804302.2804309 (2015).

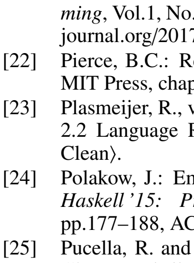
[25] Pucella, R. and Tov, J.A.: Haskell Session Types with (Almost) No Class, *Haskell '08: Proc. 1st ACM SIGPLAN Symposium on Haskell*, pp.25–36, ACM (online), DOI: 10.1145/1411286.1411290 (2008).

[26] Rust project developers: The Rust Programming Language, available from ⟨https://www.rust-lang.org/⟩.

[27] Trono, J.A.: A New Exercise in Concurrency, *SIGCSE Bull.*, Vol.26, No.3, pp.8–10 (online), DOI: 10.1145/187387.187391 (1994).

[28] Wadler, P.: Linear types can change the world!, *IFIP TC2 Working Conference on Programming Concepts and Methods* (1990), available from ⟨https://homepages.inf.ed.ac.uk/wadler/topics/linear-logic.html#linear-types⟩.

[29] Wadler, P.: The essence of functional programming, *POPL '92: Proc. 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.1–14, ACM (online), DOI: 10.1145/143165.143169 (1992).



Keigo Imai received his Doctor of Information Science degree from Nagoya University in 2012. He was at the Center for Embedded Computing Systems at Nagoya University (2009–2010); IT Planning, Inc. (2010–2013), and Research Administration Office at Kyoto University (2013–2016); Since September 2016, he has been an Assistant Professor at Gifu University. His research interests include concurrency theory, type theory and software development using functional programming languages.

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.