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 [10], albeit in a limited way.

The contribution of this paper is to extend the linear type en-

---

1. Polakow [24] and other authors also use a parameterised monad that encodes the pre- and post-conditions on linearity (See § 6).
coding 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 §2, we introduce an example of programming with linear types. §3 shows our linear type encoding using a parameterised monad and lenses in OCaml. In §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 §5. §6 introduces related work, and §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, ...). In §3, we introduce an example of programming using Safeio, in §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 §5. §6 introduces related work, and §7 concludes this paper.

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 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 [160; 280; 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 160, 280, 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 [160; 280; 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 §3.1, we introduce a framework to statically track the generation and consumption of a single linear resource. Next, in §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. §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 Figure 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 §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
val 'a lin = Lin_ of 'a
val 'a data = Data of 'a
val (>>>) : ('pre, 'pre, 'a data) monad
  -> ('a -> ('mid, 'post, 'b) monad) monad
  -> ('pre, 'post, 'b) monad
val (>>) : ('pre, 'mid, 'a data) monad
  -> ('mid, 'post, 'b) monad
  -> ('pre, 'post, 'b) monad
val 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 § 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.\footnote{However, within the framework described in § 3, it is not harmful to use Lin_ since there are no means to take linear values out of monadic variables.}

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 § 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.\footnote{The operator precedence is as follows: $e_1 >> (fun x -> e_2)$.} 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 $m = e_2 in e_1 >> (fun _ -> m)$.\footnote{The let-binding is required because OCaml is not pure, and the expression $e_2$ may have side effects.}

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>let $ex1 = unit -&gt; (empty, empty, unit data) monad</td>
<td></td>
</tr>
</tbody>
</table>
| let $m = alloc [100; 200; 300] >> map string_of_int >> update 1 "Hello" >> lookup 1 >> (fun x -> dealloc >> (print_endline x; return ()
| let $ex1; x1 = 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.\footnote{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 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>let $arr1; x1 = lookup i arr1 in</td>
<td></td>
</tr>
<tr>
<td>let $arr2; x2 = lookup i arr2 in</td>
<td></td>
</tr>
<tr>
<td>x1 + x2</td>
<td></td>
</tr>
</tbody>
</table>

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

}\]
type ('a, 'b, 'c, 'd) lens =
  {get: 'd1 -> 'a; put: 'd1 -> 'b -> 'd2}
val _0 : ('a, 'b, 'a, 'x, 'b, 'x) lens
let _0 =
  {get = (fun (a,_) => a);
   put = (fun (_,xs) b => (b, xs))}
val _1 : ('a, 'b, 'x1 * ('a, 'x) lens
let _1 =
  {get = (fun (_,a,_) => a);
   put = (fun (x,_,xs) b => x, (b, xs))}
val _2 : ('a, 'b, 'x1 * ('x2 * ('a, 'x) lens
let _2 =
  {get = (fun (_,a,_,_) => a);
   put = (fun (x,y,_,_,xs) b => x, (y, (b, xs)))}
val succ : ('a, 'b, 'x, 'y) lens
  -> ('a, 'b, ('x * 'y)s), ('x * 'y)s) lens
let succ 1 =
  {get = (fun _,xs => 1, get xs);
   put = (fun (x,_,xs) b => (x, 1, put xs b))}

Fig. 3 Lenses for manipulating slots

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 empty \(* (empty * (empty * \ldots))\) sequence as its head. The following code

\[
\begin{align*}
_0 &= Empty, _0 \in\text{empty}\times\text{empty} \\
_1 &= Empty, _1 \in\text{empty}\times\text{empty}
\end{align*}
\]

returns the view of type \(\text{empty * (empty * \ldots)}\). 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 §3.2.4.

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 \(\text{empty * 't as 't}\) is defined as follows:

\[
\begin{align*}
\text{val all_empty = empty * 't as 't} \\
\text{val run': (unit -> (all_empty, all_empty, 'a data) monad)} \\
\end{align*}
\]

\[
\begin{align*}
\text{val } & \text{run 0} : ('p, 'q, 'a) monad \\
& \to ('p, 'q, 'pre, 'post) lens \\
& \to ('pre, 'post, 'a) monad \\
\end{align*}
\]

Fig. 4 An operator for slot update in the LinMonad

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 \(\text{a}\) from the source \(\text{d1}\). On the other hand, the function put functionally updates the source \(\text{d1}\) to the type \(\text{d2}\) by writing back a value of type \(\text{b}\).

- 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.

\[\text{val run': (unit -> (all_empty, all_empty, 'a data) monad)}\]

\[\to ('a, 'b) monad\]

\[\to ('p, 'q, 'pre, 'post) lens\]

\[\to ('pre, 'post, 'a) monad\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]
an element in the slot sequence using a command. Expression
\( m \bowtie 1 \) executes the computation \( m \) in the slot referenced by lens
1. The type signature reads as follows:

- From the pre-condition ‘pre of \( m \bowtie 1 \), 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.
- Again by lens 1, ‘q is written back to ‘post and it becomes the post-condition of \( m \bowtie 1 \).

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 each element of the same index in 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.

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

By using the \( \bowtie \) 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 \( \text{lookup} i \_0 \_1 \) and writes back the sum by calling update \( i (x + y) \_0 \_1 \).

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”.

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

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.

```ocaml
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.

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

*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.
3.2.4 Polymorphic lenses using GADTs

As mentioned in § 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 Figure 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 that refers to the next element of l which is equivalent to succ 1. 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.

1get and 1put 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 (Figure 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 n in the environment obtained by applying the lens 1 to pre, and it updates the slot sequence with that lens.

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.

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 %lin for pattern matching against such linear result values.9 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 %#t, which assigns the matched linear value into an empty slot referred to by t.10

Here, we show an example of array manipulation using lens patterns. To compare programming with lens patterns with the style of § 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 Figure 1 in § 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, r) 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 § 2, which directly handles linear values.

Example 4.2 (Lens patterns matched against arrays)

9 Here, Xlin 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 Ft, 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 Figure 1).
type (_._,_._,_._,_._) lens =
  | Fst : ('a,'b,'a * 'xs, 'b * 'xs) lens
  | Next : ('a,'b,'xs,'ys) lens -> ('a,'b,'x * 'xs, 'x * 'ys) lens
  | Any : ('di -> 'a) * ('di -> 'b -> 'd2) -> ('a, 'b, 'di, '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
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 map f l =
  let rec run f =
    match f () Empty with
    | Empty, Data a -> a
    | All_emp, Data a -> a
  in
  let run f =
    match f () Empty with
    | Empty, Data a -> a
    | All_emp, Data a -> a
  in
  let alloc : 'a list -> ('pre, 'pre, 'a larr) monad
  let alloc xs = fun pre ->
    pre, Lin__ (Array.of_list pre)
  in
  val alloc : 'a list ->
    ('pre, 'pre, 'a larr) monad
let dealloc :
  ('a larr, empty, 'pre, 'post) lens ->
  ('pre, 'post, unit data) monad
let dealloc l = fun pre ->
  l.put Empty Data ()

val lookup : int ->
  ('a larr, empty, 'pre, 'post) lens ->
  ('pre, 'post, Data arr) monad
let lookup i l = fun pre ->
  let ((Lin__ arr) as arr#) = l.get pre in
  l.put Empty, Lin__ (arr#, 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 arr#) = l.get pre in
  l.put Empty, Lin__ (Array.map f arr)

Fig. 7 An implementation of LinMonad

Fig. 8 An implementation of linearly typed arrays
4.2 A semantics for lens patterns

We give the semantics of the lens pattern and %lin by macro expansion. First, let us consider the case where %lin contains a lens pattern only and generalise it for structured patterns.

Expressions with lens patterns

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

are treated as an abbreviation of the following expression:

\[
e_1 >> \text{fun } \text{lin } \# l \rightarrow e_2
\]

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

The function \text{fun } \text{lin } \# l \rightarrow e_2 is expanded to

\[
\text{Bind}_. \left( \text{fun } t m p \rightarrow _\text{put } l t m p >> e_2 \right)
\]

by the preprocessor, where \text{tmp} is a fresh variable. \text{Bind}_. is a constructor that distinguishes a \text{fun } \text{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{lin } function. The expression _\text{put } l v is a command to store the value \textit{v} in the slot pointed to by lens \textit{l}. The type of the syntax extension \text{lin } has the following form:

\[
(\text{fun } \text{lin } \# l \rightarrow e_2) :
\]

\[
(\alpha \text{ lin } \rightarrow (\text{pre}, \text{post}, \beta) \text{ monad}) \text{ bind}
\]

where \textit{\alpha}, \textit{pre}, \textit{post}, and \textit{\beta} are determined by the type of lens \textit{l} and by expression \textit{e}_2. For example, an expression with lens _\textit{\emptyset} would have the following type:

\[
(\text{fun } \text{lin } \# \emptyset \rightarrow \text{return } ()) :
\]

\[
(\text{\alpha lin } \rightarrow (\text{empty } * \text{pre}, \text{\alpha lin } * \text{pre }, \text{unit data}) \text{ monad}) \text{ bind}
\]

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

As a whole, the syntax extension let %lin \# l = e_1 in e_2 is expanded to \textit{e}_1 >> \text{Bind}_.(\text{fun } \text{tmp } \rightarrow _\text{put } l \text{tmp } >> e_2). Because the variable \text{tmp} bound to the linear result of \textit{e}_1 is immediately assigned to slot \textit{l}, and because \text{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 \text{Bind}_. on its rhs. \text{Function } _\text{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 \textit{\alpha 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{fun } %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 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 Data constructor, like data \textit{x}.
- Variable patterns outside Data are implicitly wrapped by Data to have type \textit{\tau data}.

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

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

- \text{fun lin } pat \rightarrow e expands to
\[
\text{Bind}_. \left( \text{fun } \text{conv}(\text{pat}) \rightarrow \text{put}(\text{pat}) >> e \right).
\]

Here, \text{conv}(\text{pat}) is a syntactic function for converting the pattern \text{pat}, which is given later. \text{put}(\text{pat}) is a function that generates a command _\text{put } l_1 \text{tmp}_1 >> \cdots >> _\text{put } l_n \text{tmp}_n for the lens \textit{l}_1, \ldots, \textit{l}_n appearing in \text{pat}, where the variables \text{tmp}_1, \ldots, \text{tmp}_n are freshly generated by \text{conv}(\text{pat}).

- \text{function lin case } | \cdots | \text{case } expands each clause \text{case } of the form \text{pat } \rightarrow e, as in \text{lin}.

- \text{let lin } pat = e_1 in e_2 is expanded similarly to the case of \textit{e}_1 >> \text{fun lin } pat \textit{e}_2.

- \text{match lin e with case } | \cdots | \text{case } expands similarly to \textit{e} >> \text{fun lin case } | \cdots | \text{case }.
let rec _traverse p =
  match p with
  | _ is a lens pattern #l
    -> Generate a fresh variable tmp;
       Record a pair of lens / 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

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._. 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 funolin syntax, the function only accepts a linear value directly from the slot sequence 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 Figure 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) 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.

Although lin and bind should be abstract types, they could not be hidden because the code generated by the syntax extension will use them.

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, _1) -> 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, _12) -> 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(c1, ..., cn) or lens references are allowed in e.
- The lens reference !i/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 !l1, ..., !ln 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 e assigns
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.

```
let rec map0 (l : 'a 'b linlist) = l.put l0 (fun a -> f a, Data () monad)
| l0 , l1 , l2 , Unit_empty , a -> l.put l0 a, Data () monad
| Nil -> put_linval _1 Nil >> ('a 'b linlist, empty) monad

val rev_map : ('a -> 'b) -> ('a 'b linlist, empty, 'pre, 'mid) lens -> ('b linlist, empty, 'pre, 'mid) lens

let rec rev_map f s1 s2 s3 s4 = rev_map f s1 s2 s3 s4 >> put_linval s4 Nil
```

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 'b linlist, empty, 'pre, 'mid) lens -> ('b linlist, empty, 'pre, 'mid) lens

let rec map' f s1 s2 = map' f s1 s2 >> put_linval s2
```

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 'b linlist, empty, 'pre, 'mid) lens -> ('b linlist, empty, 'pre, 'mid) lens

let rec map f s1 s2 s3 s4 = map f s1 s2 s3 s4 >> put_linval s4 [Xfixret 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

let rec ex5 () =
| Cons(x, #_0) ->
| cut_monad
| Nil -> return ()

in
| Cons(Data(f x), !!_1) >>
| loop ()
```

□

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 (0, τ) recv denotes receiving a value of type τ before behaving as session 0, (0, τ) send denotes sending a value of type τ before behaving as session 0, 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
```

□

Figure 14 shows the signature of a communication API based on session types. The type (θ1, θ2) 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

---

13 This type represents communication steps in right-to-left order owing to the postfix syntax of OCaml types.
Santa Claus is waiting in the main thread by using accept.

We quote the problem from [27]:

programming proposed by Trono, and it has served as a benchmark that initially sends a value from the server to the client using the API in Figure 15. For example, s2c finish may occur and the types of messages at each peer coincide.

communication on that channel will be consistent (i.e. no dead-

accept s is the one that the \(2\) request using

deer and elf a thread and let them establish a session with Santa by

dynamic 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 \(\theta\ s\ list\) represents a list of sessions. Although similar to the linearly typed list in Figure 12, type \(s\ list\) differs from it in that it holds linearly typed content of type \(\theta\ lin\).

Function \(\text{iter}\) is only used in the form \(\text{iter} i l1 l2 l3 f\) =

\[
\text{if } i = 0 \text{ then }
\begin{align*}
\text{return } ()
\end{align*}
\]

\[
\text{else }
\begin{align*}
\text{match }& \text{get_lin } l1 \text{ with }
\begin{cases}
| S\text{Cons}(_, _, _) & \to f () \to \text{iter } (i-1) l1 l2 l3 f
| SNil & \to \text{put_linval } l3 \text{ SNil}
\end{cases}
\end{align*}
\]

Fig. 16 Iteration on a list of sessions

cate 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 \(\theta\ s\ list\) represents a list of sessions. Although similar to the linearly typed list in Figure 12, type \(s\ list\) differs from it in that it holds linearly typed content of type \(\theta\ lin\).

Function \(\text{iter}\) is only used in the form \(\text{iter} i l1 l2 l3 f\) =

\[
\text{if } i = 0 \text{ then }
\begin{align*}
\text{return } ()
\end{align*}
\]

\[
\text{else }
\begin{align*}
\text{match }& \text{get_lin } l1 \text{ with }
\begin{cases}
| S\text{Cons}(_, _, _) & \to f () \to \text{iter } (i-1) l1 l2 l3 f
| SNil & \to \text{put_linval } l3 \text{ SNil}
\end{cases}
\end{align*}
\]

Fig. 16 Iteration on a list of sessions

cate 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 \(\theta\ s\ list\) represents a list of sessions. Although similar to the linearly typed list in Figure 12, type \(s\ list\) differs from it in that it holds linearly typed content of type \(\theta\ lin\).

Function \(\text{iter}\) is only used in the form \(\text{iter} i l1 l2 l3 f\) =

\[
\text{if } i = 0 \text{ then }
\begin{align*}
\text{return } ()
\end{align*}
\]

\[
\text{else }
\begin{align*}
\text{match }& \text{get_lin } l1 \text{ with }
\begin{cases}
| S\text{Cons}(_, _, _) & \to f () \to \text{iter } (i-1) l1 l2 l3 f
| SNil & \to \text{put_linval } l3 \text{ SNil}
\end{cases}
\end{align*}
\]

Fig. 16 Iteration on a list of sessions

cate 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 \(\theta\ s\ list\) represents a list of sessions. Although similar to the linearly typed list in Figure 12, type \(s\ list\) differs from it in that it holds linearly typed content of type \(\theta\ lin\).

Function \(\text{iter}\) is only used in the form \(\text{iter} i l1 l2 l3 f\) =

\[
\text{if } i = 0 \text{ then }
\begin{align*}
\text{return } ()
\end{align*}
\]

\[
\text{else }
\begin{align*}
\text{match }& \text{get_lin } l1 \text{ with }
\begin{cases}
| S\text{Cons}(_, _, _) & \to f () \to \text{iter } (i-1) l1 l2 l3 f
| SNil & \to \text{put_linval } l3 \text{ SNil}
\end{cases}
\end{align*}
\]

Fig. 16 Iteration on a list of sessions

cate 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 \(\theta\ s\ list\) represents a list of sessions. Although similar to the linearly typed list in Figure 12, type \(s\ list\) differs from it in that it holds linearly typed content of type \(\theta\ lin\).

Function \(\text{iter}\) is only used in the form \(\text{iter} i l1 l2 l3 f\) =

\[
\text{if } i = 0 \text{ then }
\begin{align*}
\text{return } ()
\end{align*}
\]

\[
\text{else }
\begin{align*}
\text{match }& \text{get_lin } l1 \text{ with }
\begin{cases}
| S\text{Cons}(_, _, _) & \to f () \to \text{iter } (i-1) l1 l2 l3 f
| SNil & \to \text{put_linval } l3 \text{ SNil}
\end{cases}
\end{align*}
\]

Fig. 16 Iteration on a list of sessions

cate 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 \(\theta\ s\ list\) represents a list of sessions. Although similar to the linearly typed list in Figure 12, type \(s\ list\) differs from it in that it holds linearly typed content of type \(\theta\ lin\).

Function \(\text{iter}\) is only used in the form \(\text{iter} i l1 l2 l3 f\) =

\[
\text{if } i = 0 \text{ then }
\begin{align*}
\text{return } ()
\end{align*}
\]

\[
\text{else }
\begin{align*}
\text{match }& \text{get_lin } l1 \text{ with }
\begin{cases}
| S\text{Cons}(_, _, _) & \to f () \to \text{iter } (i-1) l1 l2 l3 f
| SNil & \to \text{put_linval } l3 \text{ SNil}
\end{cases}
\end{align*}
\]

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 [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 communi-
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) =
  match s with
  | #_0, Elf ->
    loop ch (ecount+1, rcount)
  | #_0, Reindeer ->
    loop ch (ecount, rcount+1))

... fun (ecount, rcount) ->
    if rcount<9 then
      loop ch (ecount,0)
    else if ecount<3 then
      loop ch (ecount-3, rcount)
    else
      loop ch (ecount, rcount)

val santa : santa_ch -> (all_empty,_,_) monad

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

rigue [6], is the first encoding of linearly typed resources by a
parameterised monad in OCaml, and it is mostly reproduced in
§ 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 \( \lambda x. y. x + y \),
which calculates the sum of two linearly typed integers, can be written
as \( \lambda n \) \((\lambda m \) \((\lambda s \) \((\lambda z \) \(z)))\) 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 abstrac-
tions 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 de-
pendingencies. 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

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 encod-
ings, 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 im-
plementation 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. How-
ever, programming with such stack manipulations becomes cum-
ersome 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 re-
sources 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 depend-
dencies.

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
\( (\lambda x. y. x + y) 21 \) by defining add in the following way:

\[
\begin{align*}
\text{val add : (int lin, empty, 'pre', 'mid') lens ->} \\
& (\text{int lin, empty, 'mid', 'post') lens ->} \\
& ('pre', 'post, int lin) monad
\end{align*}
\]

and by passing parameters via lenses as follows:

\[
\begin{align*}
([\text{linret} \ 42]) & \Rightarrow \text{fun} \text{lin}_{\ _0} \Rightarrow \\
([\text{linret} \ 21]) & \Rightarrow \text{fun} \text{lin}_{\ _1} \Rightarrow \\
& \text{add}_{\ _0 \ _1}
\end{align*}
\]

However, we have not considered how to introduce higher order
functions such as \( \lambda f. f x \). Since \text{fun} \text{lin} cannot be nested like
in \text{fun} \text{lin}_{\ _1} \Rightarrow \text{fun} \text{lin}_{\ _2} \Rightarrow \), and because linocaml
stores linear values in slots rather than in variables, it is not obvi-
ous how to encode such curried functions.\footnote{\text{For example, by using extend and shrink in §3.2.3, we can construct}}
On the other hand, by using the host language abstraction mechanism, we can construct higher order functions by slot manipulation and lens passing, like `iterM`, `mapM`, as we have seen in § 3.2.2 (Example 3.3 and Example 3.4). Furthermore, in Example 3.3, the function passed to `iterM` updates the linear values by passing them via slot `_1`:

```haskell
iterM (fun i x ->
  lookup i 0 @> _1 >>
  fun y ->
  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 (\( \text{fun i x -> ..} \)) and (\( \text{fun}\text{lin} \ #_0 \to ..) \) in \text{linocaml} do not have this limitation and can be used freely. Thus, although comparison of the expressiveness is not evident, the authors expect \text{linocaml} to be as expressive as the linear lambda calculus.

7. Conclusion
This paper described an encoding of linear types in \text{OCaml} using a parameterised monad and lenses, which we have made available through the \text{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 \text{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 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 such as Rust being the only widely known programming language supporting them natively. By introducing them in \text{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 \text{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 \((a\to b)\to (a\to xs), (b\to xs)\) \text{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


© 2014 Information Processing Society of Japan

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.