IPSJ Transactions on Programming Vol.0 No.0 1–14 (??? 2007)

Regular Paper

Lightweight linearly-typed programming with lenses and monads

Keigo Imai[1,a)] Jacques Garrigue[2,b)]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

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, thus they are used to check behavioural aspects of programs like resource accesses 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 like 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. はじめに

線形型は，値がただ一度だけ使用されることを保証する型であり，リソースの管理や通信プロトコルといったプログラムの詳細な振舞いを静的に検査できる．

線形型に類似した型機能がプログラミング言語に取り入れられた例は，Rust [26] のアフィン型，Clean [23] の一意型 (uniqueness types) 等がある．さらに，近年，Haskell [5] の処理系 Glasgow Haskell Compiler においても線形型の導入が提案されている [4]．特に Rust においては，変数の出現をアフィン型を用いて静的に追跡することにより，効率のよいメモリ管理を実現している．

しかしながら，線形型を既存の汎用的なプログラミング言語に導入するには，コンパイラや型システム全体に渡って修正が必要となり，実装のコストが大きい．線形型を既存の言語機能を用いてエンコードし，プログラミング言語のライブラリの形で導入できれば，その言語におけるリソース管理の信頼性と効率性を向上させることができるだけでなく，既存の言語機能との相乗効果により，様々なプログラミングテクニックの発展が期待できる．

Haskell におけるエンコーディング 線形型のプログラミング言語におけるエンコーディングは，Haskell の上において発展してきた．特に Polakow [24] によるエンコーディングは，HOAS (Higher Order Abstract Syntax) [20] の形で線形ラムダ計算を直接的に埋め込んでおり，利用しやすいように見える．しかしこれは Haskell の型クラスとパラメタ間依存 (functional dependencies) [13] を用いて変数の消費を追跡しており，他のプログラミング言語への応用は難しい．

本論文は，OCaml において，**パラメタ化モナド** [1] と**レンズ** [6], [21] を用いた線形型のエンコーディングを示し，これを用いたライブラリ `linocaml` を提案する．ここで，パラメタ化モナドは計算の前条件と後条件を表す型パラメータをもつモナドであり，線形なリソースの生成や消費を静的に表現する[*1]．レンズは，データ型におけるある一点を表す抽象であり，前条件や後条件を表すデータ型と組み合わせることで，線形なリソースの間接的な参照として扱う．

本論文で提示する線形型のエンコーディングは，多くのプログラミング言語で利用可能なパラメタ多相のみを用いており，線形型の実装手法として軽量かつ高い可搬性を持つのが長所である．

更に，我々は OCaml の構文拡張機構を用いてパターンマッチ構文を拡張し，線形型をもつリスト等の構造化されたデータの操作方法を与える．この構文拡張により，線形型を用いたより柔軟なプログラミングが可能になる．応用の例として，セッション型 [9] のエンコーディングを導入し，線形型をもつデータ構造のパターンマッチを併用することで，並行プログラミングにおけるサンタクロース問題 [2], [27] の解法を示す．

著者らの先行研究 本論文は，Garrigue が OCaml メーリングリスト [7] にて示した `Safeio` と，Imai らが [10], [11] で発表した，パラメタ化モナドとレンズを用いたセッション型のエンコーディングの発展である．本論文で導入する構文拡張のうち，線形型に対するパターンマッチは [11] においても限定的に導入している．

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

[*1] 前述の Polakow のエンコーディングも，計算の前条件と後条件を表す型を用いており，パラメタ化モナドの一種とみなすことができる．

© 2007 Information Processing Society of Japan

本論文の貢献は，Safeioやセッション型実装における線形型の扱いを整理し，配列や，リストなどの構造化されたデータを扱う関数プログラミングへと拡張したことである．

本論文の構成 § 2 で，線形型によるプログラミングの例を導入する．§ 3 で，OCaml におけるパラメタ化モナドとレンズを用いた線形型のエンコードを示す．§ 4 で，線形型をもつ構造化データを扱うための構文拡張を導入する．応用例として，§ 5 で，セッション型のエンコードと，線形型をもつ構造化データを活用した，サンタクロース問題の解法を示す．§ 6 で関連研究を紹介し，§ 7 で，本論文の結論を述べる．

ソースコード linocaml のソースコードは，https://github.com/keigoi/linocaml で公開している．

2. 線形型とリソース管理

線形型によるプログラミングを，Wadler の [28] における例を用いて概観する．本節では，線形型を備えた仮想的なプログラミング言語における，純粋関数的な配列の API を考える．構文と型は OCaml の記法を用いる．すなわち，型変数は 'a，'b，... のようにクォートが付き，型コンストラクタは 'a list ('a 型のリスト) のように後置する．

Example 2.1 (線形型をもつ配列の API) 線形型をもつ配列の API を次に与える．ここで 'a **larr** は要素の型が 'a である線形型の配列型である．

```
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
```

それぞれの操作の意味は次の通りである．

- alloc xs は，リスト xs を内容とする配列を返す．
- dealloc arr は，配列 arr のメモリ領域を解放する．
- lookup i arr は，arr それ自身と，配列 arr の i 番目の要素の対を返す (ここでは簡単のため配列の境界外アクセスは考慮しない)．
- update i e arr は，配列 arr の i 番目の要素を e に置き換えた配列を返す．
- map f arr は，配列 arr の全要素を f に適用して更新した結果を返す．
- to_list arr は，arr それ自身と，配列 arr の内容をリストに変換した値の対を返す．

線形型は，その値がただ 1 回だけ使用されることを保証するため，実際に確保される配列のメモリ領域は再利用できる．すなわち，純粋関数的なインタフェースを持つにも関わらず，更新操作の内部実装では配列を破壊的に更新することで，配列のメモリ確保のコストを低減できる． □

線形型を用いたとき，メモリ領域に割り当てられた型を更新できるのは特徴的である．

Example 2.2 (型の更新) 配列の型を更新する例として，次のプログラムを考える．

```
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
delloc arr3;
print_endline x
```

このプログラムは，int **larr** 型の整数の配列 100，200，300 を確保し，関数 string_of_int により各要素を文字列に変換して string **larr** 型の文字列の配列へと更新 (map) する．そして 1 番目の要素を"Hello" に更新 (update) し，更新した箇所を参照 (lookup) して変数 x に束縛し，標準出力に印字にする．ここで線形型は，変数 arr，arr1, arr2, arr3 が 1 度しか消費されないことを保証する．このため，map や update による純粋関数的な更新操作について，実際には同じメモリ領域を破壊的に更新する効率のよい実装が可能である[*2]． □

線形型を利用したライブラリでは，線形性に違反するとプログラムの安全性が失われる．例えば

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

は，最初の配列 arrを，mapによって (型を string に) 更新した後に再び int で書き込むことになり，安全でない．このような線形値のアクセスは，線形型システムの枠組みでは静的に排除できる．

3. パラメタ化モナドとレンズによる線形型のエンコード

OCaml の型システムは変数を使う回数を追跡しないため，例 2.2 の arr, arr1, arr2, arr3 のように線形型の値を変数束縛できると，線形型の制限を破るおそれがある．このことから我々は，線形型の値を変数に直接束縛せず**暗黙に**扱う計算の体系として，パラメタ化モナド **LinMonad** に基づくコンビネータによるプログラミングの方法を与える．このパラメタ化モナドは，型において線形な値の使用が陽に現れるため，線形性の制約を OCaml の型システムで静的に保証できる．

まず，§ 3.1 で単一の線形リソースの生成と消費を静的に追跡する枠組みを導入する．次に § 3.2 で，前条件と後条件を複数のデータを保持するスロット列で拡張するとともに，**レンズ**による間接参照により，複数の線形リソースを扱える枠組みへと発展させる．§ 3.3 で，状態モナドによる実現とともに，具体的な線形型 API の実装技法を示す．

3.1 パラメタ化モナド

LinMonad を用いたプログラミングの典型的な題材として，図 1 に，例 2.1 で示した線形型付きの配列操作の API を示す．各関数が返すのは配列ではなく配列操作を表す**モナド値**（コマンド）である．モナド値は型（*pre*，*post*，α）monad を持つ．モナド値は UNIX のコマンドのように連接でき，直前のコマンドから型 *pre* の線形値を入力し，次のコマンドへ型 *post* の線形値を出力するとともに，型 α の計算結果を返す．

型'a **larr** は，線形型を区別するコンストラクタ **lin** を用いて，線形型を持つ配列 'a array **lin** のエイリアスとして宣言する．計算結果の型は制限のない (線形型でない) 型を表す **data** でラップする．**lin** と **data** はそれぞれ線形型と制限のない (線形型でない) 型を表すコンストラクタであり，本節では特に役割を持たないが，§ 4 のパターンマッチにおいて重要な役割を果たす．

例えば，配列を生成する alloc *xs* :（empty，'a **larr**，unit **data**）monad は入力に型 empty の空値をとり，配列 'a

---

[*2] ただし OCaml の配列型は，浮動小数点数の配列型 float array において特殊化 (アンボックス化) されており [16]，他の配列型とメモリ表現の互換性がないため，実装には少し工夫が必要である (§ 3.3)．

© 2007 Information Processing Society of Japan

```
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 LinMonad を用いた配列操作 API
Fig. 1 An 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 パラメタ化モナド LinMonad
Fig. 2 A parameterised monad LinMonad

larr を確保して出力するとともに，計算結果として型 unit のユニット値を返す．その反対に，配列を廃棄する dealloc は廃棄する配列を入力に取り，出力は空値である．配列の i 番目の要素を参照する lookup i : ('a larr, 'a larr, 'a data) monad は，入力した型 'a larr の配列を変更せずそのまま出力すると同時に，型 'a の要素を返す．map や to_list も同様に例 2.1 と対応している．

LinMonad のシグネチャを図 2 で導入する．型'a lin, 'a data は，それぞれ単一のコンストラクタ Lin__, Data で値を単にラップする．Lin__ は線形型の API 実装 (§ 3.3) において用いるが，LinMonad のエンドユーザーは使ってはならない[*3]．

return は入力を消費せず，次の操作にそのまま出力する「純粋」なコマンドであり，前条件と後条件が同一の型 'pre である．§ 4 における拡張において一貫性を保つため，return の結果の型は 'a data のように型コンストラクタ data でラップするが，次に示すバインド演算 >>= や run において data は取り除かれる．

線形性のうち，線形値が捨てられない性質は，コマンドを連接するバインド演算 >>= の型付けと，後に示す run によって保証される．バインド演算は UNIX のパイプに相当し，左辺のコマンドの計算結果 ('a) を右辺の関数に渡すとともに，左辺から出力された線形値 ('mid) を右辺のコマンドへ入力する．型シグネチャは，計算結果の値からコンストラクタ data を取り除いて右辺の関数に渡すだけでなく，左辺の出力と右辺の入力が型 'mid で整合することを要求している．このことにより，左辺の出力が線形値 (型 'a lin) であるにもかかわらず，右辺が入力を捨てる empty である場合は，型エラーとして静的に検出される．バインド演算によって合成されたモナド値は，型 'pre の入力を左辺のコマンドに与え，右辺の型 'post の出力を出力するとともに，右辺の計算結果 ('b) を返す．

モナド値の計算結果は，e_1 >>= **fun** x -> e_2 [*4]のように，>>= の右辺における関数の仮引数に束縛し，続きの計算で利用できる．e_1 >> e_2 は，このように左手の計算の結果の値を捨てるバインド操作 e_1 >>= **fun** _ -> e_2 とほぼ同じ意味をもつ[*5]．

線形性のもう一つの性質である，線形値が複製されない性質は，コマンドの入力や出力が変数束縛されず，バインドを介して暗黙に渡されることによって保証される．

空値はコンストラクタ Empty で表す．コマンドは関数 run を介して実行される．出力の型が empty であることにより，最後のコマンドが線形値を出力しないことを保証する．

Example 3.1 (パラメタ化モナドによる配列操作) 次のプログラムは，例 2.2 をパラメタ化モナドを用いて模倣したプログラムである．

```
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
```

この例において，alloc によって生成された配列は，map, update, lookup の各コマンドを経て，dealloc で破棄される．全体として，関数 ex1 は入力と出力が空 (empty) であり，実行の過程で配列を例 2.2 の通り操作するコマンドを返す[*6]．
□

3.2 レンズによる複数のリソースの扱い

LinMonad において複数の線形値を同時に扱う枠組みを導入する．例えば，2 つの配列の要素の和を計算する

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

のような操作を書けるようにする．アイデアは，パラメタ化モナドのコマンドの入力と出力において複数の線形なリソースを保持する**スロット列** [7], [10], [11] というデータ構造を用いることである．そして，各リソースへの線形でないアクセスをこれまで通り制限しつつ，**レンズ**で間接的に参照可能にすることにより，線形性を保証する枠組みを構築する．

3.2.1 準備

スロット列 複数のリソースを保持するデータ構造として，スロット列を与える．スロット列は，(x_0, (x_1, (x_2, ...))) のように 2 番目の要素でネストした対で構成されるデータ構造である．特に，線形なリソースを含まないスロット列の型

---

*3 しかしながら，本節 (§ 3) までの枠組みでは線形値を陽に束縛する方法はないため，エンドユーザーが Lin__ を用いても特に害はない．

*4 構文の結合は e_1 >>= (**fun** x ->e_2) のようになる．

*5 ただし，OCaml の値呼び評価のため，e_2 が (print 文など) 副作用を持つ式の場合，>> の呼び出しの前にその副作用が生じる．

*6 (print_endline x; return ()) は ; の左辺の print_endline が印字した後に返す () を破棄し，return () の (何もしない) コマンド自体を返すことに注意する．

© 2007 Information Processing Society of Japan

```
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 スロット列を操作するレンズ
Fig. 3 Lenses for manipulating slots

を，empty * (empty * (empty * ..)) という無限列の型で表す．このような無限木をもつ型を OCaml コンパイラで利用するには，2 つの方法がある．

(1) OCaml の多相ヴァリアントかオブジェクト型は，同値再帰型 [22] を構成できる．リストのコンスを多相ヴァリアントのコンストラクタ `cons で表現すると，[`cons **of** empty * [`cons **of** empty * [`cons **of** empty * ..]]] と無限に続く型は [`cons **of** empty * 't] **as** 't と書ける．ここで T **as** 't は，T に出現する型変数 't をそれ自身で置き換えた型 T[(T **as** 't)/'t] と等しい同値再帰型である．オブジェクト型のメソッドでコンスを表現すると <cons : empty * 't> **as** 't となる．

(2) OCaml コンパイラの -rectypes 拡張を用いれば，任意の型コンストラクタについて同値再帰型の構成が許されるようになり，empty * (empty * (empty * ..)) をより直接的に empty * 't **as** 't と書ける．

本論文では見た目の簡単さのため後者を用いる．このような無限木をもつ型は OCaml 以外の一般のプログラミング言語では利用できないが，同様のことは再帰型の有限な展開を用いて模倣できる (§ 3.2.3)．

レンズ レンズ [6] は，双方向プログラミングの文脈における双方向変換の抽象化であり，あるデータ構造から別のデータ構造（ビュー）への変換と，ビューに施した変更を元のデータ構造に書き戻す逆方向の変換を束ねたものである．さらに，Haskell の lens ライブラリ [15] や Pickering ら [21] のレンズは，異なる型の値への変更についても書き戻しを許している．

レンズの型定義と，スロット操作のためのレンズを図 3 に示す．構成要素は次の通りである．

- レンズは，ビュー関数 get と書き戻し関数 put の対で表される．ここで型パラメータ 'd1, 'd2 はレンズが参照する複合的なデータの型を表す．フィールド get により，レンズが参照する部分の型 'a のビューを取り出す．一方，フィールド put は，型 'd1 のデータのうちレンズが参照する部分に型 'b の値を書き戻し (関数的に更新し)，

```
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 スロット列を更新する演算子
Fig. 4 An operator for slot update in the LinMonad

型 'd2 のデータを返す．

- レンズにより，スロット列の各要素の線形性を維持しつつ，スロット列の任意の有限な位置の要素を間接的に参照できるようになる．スロット列の 0 番目の要素を参照するレンズ _0 を[*7]，対の左側の要素を参照するレンズとして導入する．同様に，スロット列の 1 番目，2 番目の要素を参照するレンズ _1, _2 を導入する．

- スロットの 3 番目以降の要素は，ある要素を参照するレンズから，次の要素を参照するレンズを生成する，関数 succ を用いて構成できる．例えば，1 番目，2 番目の要素を参照するレンズはそれぞれ succ _0, succ (succ _0) と書ける．ただし OCaml の値多相の制限により，このような式は単相的になり，複数の型を扱うことができない．この制限は，§ 3.2.4 で示すように，レンズの実体を GADT により構成することで回避できる．

Example 3.2 (レンズによるスロット操作) スロット列に対するレンズの直観的な振舞いを紹介する．empty 型の唯一のコンストラクタを Empty とすると，線形型の値を持たない，型 empty * 't **as** 't を持つスロット列 all_empty は

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

と定義できる．これは，リストの頭に Empty を持つ循環的なリストである．ここで，0 番目の要素に配列 arr : int **larr** を割り当てるには，_0 を用いて次のようにする．

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

さらに，2 番目の要素に arr1 : string を割り当てるには次のようにする．

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

□

3.2.2 レンズによる複数リソースの操作

スロット列の型と，モナド値の計算を実行する関数 run' と，レンズが参照するスロットの要素をコマンドで更新する演算子 @> を図 4 の通り導入する．m @> l は，線形型を扱う計算 m をレンズ l が参照するスロットにおいて作用させる．型シグネチャは，次のことを説明している．

- m @> l の前条件 'pre から，レンズ l が参照する，型 'p を持つ線形なリソースが取り出される．

- リソース 'p は，計算 m において消費され，m の後条件 'q と結果の値 'a が返される．

[*7] スロット列の最初の要素を 0 番目と数える．

© 2007 Information Processing Society of Japan

- レンズ l によって，'q が 'post に書き戻され，m @> l の後条件となる．

次の例で，レンズを用いて複数の線形なリソースを扱う．ここで，iteriM [*8] と，map の変種 mapiM f l を用いる．iteriM f l は，レンズ l の位置にある配列のそれぞれ i番目の要素 e_i について，関数 f i e_i を呼ぶ．

Example 3.3 (レンズを用いた複数の線形リソースの操作 (1)) 複数の線形なリソースを用いる例として，複数の配列の和を計算する例を考える．次の例は配列 23，34，45 と 100，200，300 の各要素それぞれの和からなる配列 123，234，345 を返す関数である．

```
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 @> _0 >>= fun xs ->
  dealloc @> _0 >>
  dealloc @> _1 >>
  return xs
```

配列を確保すると，演算子 @> により，それぞれの配列を 0 番目と 1 番目のスロットに割り当てる．次に，iteriM 関数により，1 番目のスロットの配列の各要素を 0 番目のスロットの配列の同じ位置の要素との和で更新し，リストに変換した値を返す．iteriM に渡された無名関数 **fun** i x -> ... は，lookup i @> _1 により 1 番目のスロットの i番目の要素を参照し，計算した和を update i (x + y) @> _1 により格納している． □

次の例は mapiM により配列の型を更新する．

Example 3.4 (レンズを用いた複数の線形リソースの操作 (2)) 次のプログラムは，配列 100，200，300 を文字列に変換し，各要素をそれぞれ別の文字列の配列 "abc"，"def"，"ghi" と連接した文字列からなる配列 "abc123"，"def234"，"ghi345" を返す．

```
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
```

□

iteriM, mapiM の型シグネチャは図 5 の通りである．型シグネチャにより，次のような線形性の制約を表現している．

- iteriM f l の第 1 引数の関数 f の型は，反復の過程で何度も呼び出されるため，環境における線形型の値の型を更新できない．このため，モナド値の型の前条件と後条件は同じ 'pre である[*9]．第 2 引数 l は型 'a **larr** の反復対象の配列を参照するレンズである．iteriM は配列の型を更新しないため，レンズの 3 番目の型引数と 4 番目の型引数は，それぞれ 1 番目の型引数 'a **larr** と 2 番目の型引数 'pre と同一である．

```
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 関数 iteriM と mapM の型シグネチャ
Fig. 5 Signatures for iteriM and mapM

- mapiM f l の第 1 引数 f は線形型の値の型を更新しないが，モナド値の結果の型として変換後の型 'b **data** 返す．その一方で，配列を参照する第 2 引数 l は，この関数が線形型 'a **larr** の値を更新して新しい配列 'b **larr** を返すことを示している．この更新を反映して，更新後の後条件の型は 'post となる．

3.2.3 同値再帰型を用いないスロット列

OCaml 以外の一般的なプログラミング言語においては，同値再帰型を用いないスロット列の表現が必要になる．このことを実現するため，スロット列をスロット 1 つ分だけ伸張する関数 extend と，短縮する関数 shrink を導入する．

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

extend と shrink を用いれば，スロット列を必要なだけ有限に展開できる．

Example 3.5 (スロット列の伸張と短縮) 例 3.4 の関数 example3 は，2 つのスロットを使うため，関数 run の入力 empty から次のように 2 つのスロットを extend で伸張すれば実行できる．run の出力は empty でなければならず，スロット列は最後に shrink で短縮する．

```
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 GADT を用いた多相的なレンズ

§3.2.1 で言及したように，succ _0 のようなレンズは多相性の値制限のため，単相的になる．このため，レンズで異なる型のデータを操作するには succ (succ _0) のようにその場でレンズを合成しなければならず，煩雑である．

レンズを 図 6 のように GADT [8] のコンストラクタで構成することにより，値制限を回避でき，合成されたレンズを

*8 関数名の最初の部分の iteri は，配列の各要素について反復 (iterate) する高階関数の，添字 (index) 付きの変種であることを表す OCaml 標準ライブラリの命名規則を真似ている．さらに末尾の M でモナド値を返す関数を受け取る変種であることを表現している．

*9 return の場合と異なり， f が返す計算は副作用を含みうる．型シグネチャで述べているのは，f が線形型の値の型を更新しないことだけである．

© 2007 Information Processing Society of Japan

も多相的に保つことができる．

Fst はスロットの 0 番目を参照するレンズであり，Next l は succ に相当する，l の次の要素を参照するレンズである．Any（*get*，*put*）は，任意のビュー関数 *get* と書き戻し関数 *put* からなるレンズである．_0，_1，_2，_3 はそれぞれコンストラクタを用いて定義されているため，単相性制限の対象とはならず，多相的である．

lget と lput はそれぞれレンズのビュー操作と書き戻し操作である．ここで型注釈における **type** a xs などは局所抽象化型 (locally abstract type) であり，GADT に対するパターンマッチにより詳細化される型を表す．

3.3 モナドと API の実装

LinMonad の状態モナドによる実装 LinMonad のシグネチャ (図 2) の状態モナド [29] による実装と，スロット列による拡張を図 7 に示す．型（'pre，'post，'a）monad は状態の型が 'pre から 'post へと変化する状態モナドであり，実体は関数型'pre -> 'post * 'a である．

モナドの操作 return，>>=，>>，run は，標準的な状態モナドに加えて，モナド値の計算結果を Data でラップした型 'a **data** で扱っている．run は，前条件と後条件が empty であることを要求するため，状態の値 Empty を明示的に授受している．run' は，Empty の代わりに all_empty を用いる以外の違いはない．@> は，環境 pre からレンズ l で取り出した値によりモナド値 m を実行し，スロット列をレンズで更新する．

線形型 API の実装 線形型 API の実装においては，型システムはリソースの線形性を保証しない．このため，API の実装ではドメインや保証したい性質に応じた技法が必要になる．例えば，効率のよい配列の実装では，配列の型を更新するために，実装において型安全でない操作が必要となる．

具体的な例として，型の更新を伴う実装を図 8 に与える．OCaml の配列型は，浮動小数点数の配列型 float array において特殊化 (アンボックス化) される [16] ため，任意の型の配列を扱うには工夫が必要である．配列の実体は，値の型を任意の型に (型安全でない方法で) 変換する次の関数を用いる．

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

Obj.t は任意の型の値を埋め込む型である．

OCaml の配列の内部表現は初期化時に渡される値によって動的に決まるため，alloc の実装において，配列を確保する Array.make には整数 0 を与え，整数型の (float型でない) 配列として確保する．配列を参照する lookup や map では，要素の参照の際に Obj.obj により Obj.t 型から実際の型に復元する．配列を更新する update や map においては，Obj.repr により Obj.t 型に変換した要素をストアする．to_list においては，型 Obj.t list のリストを Obj.magic で真の型に変換する．

各 API は，配列を線形に扱い，参照を増やす（例えばグローバル変数に格納する）ような操作を行わないため，全体として線形なアクセスを保証している．

4. 構造化データを扱うための拡張

本節で，線形型をもつ構造化されたデータに対するパターンマッチの構文拡張を示す．パターンマッチは関数型プログラミングの強力な機能であり，再帰的なデータ構造と組み合わせると，多様なアルゴリズムを記述できる．特に，線形型付きのリストは，任意の数の線形なリソースを動的に扱えるようになるため，重要である．

4.1 線形値のパターンマッチのための API デザイン

線形型の値に対するパターンマッチのため，パラメタ化モナドの結果の型を（'pre，'post，'a **lin**）monad のように線形型を含むよう拡張する．

この線形型をもつ結果の値に対するパターンマッチのための構文拡張 **let%lin** を導入する[*10]．**let%lin** *pat* = e_1 **in** e_2 は，モナド式 e_1 の結果の値をパターン *pat* に束縛し，モナド式 e_2 を実行する．ここで，*pat* は線形な値を空のスロットに割り当てる**レンズパターン#l** で拡張する[*11]．

レンズパターンによる配列操作の例を示す．本節のレンズパターンと，§ 3 におけるパターンマッチがないプログラミングの対比のため，まず，配列操作の結果の型を（'pre，'post，'a **lin**）monad の形に変更した API を与える．

Example 4.1 (配列 API の変更) 新しい配列 API を図 9 のように決める．§ 3 の 図 1 の配列 API との違いは次の通りである．

- 配列を返す（dealloc 以外の）関数は，結果の型が τ **lin** の形であり[*12]，線形型をもつ更新後の配列（と，lookup においては参照した値の対）を返すことを直接的に表している．
- alloc 以外の関数は，操作対象の配列を参照するレンズの型（'a **larr**，empty，'pre，'post）lens を引数に取る．この型は，配列 API がレンズの引数で参照する配列を消費するという線形型の性質を表している．つまり，スロット列 'pre のうち，レンズが参照する配列 'a **larr** を消費し，そのスロットを空 (empty) にして 'post に書き戻す．関数の戻り値の型は（'pre，'post，τ）monad であり，前条件 'pre と後条件 'post はレンズによって操作したデータ構造の型そのままである．
- lookup の結果の型（'a **larr** * 'a **data**）**lin** のように，無制限の (線形型でない) 値 'a を返す場合は，後のパターンマッチにおけるパターンの区別のため，'a **data** のようにラップする．

□

レンズパターンを用いたプログラミングは，§ 2 で示した線形型によるプログラミングに近い，より直観的な形になる．

Example 4.2 (レンズパターンを用いた配列操作) レンズパターンと図 9 の配列 API を用いると，例 2.2 の配列操作は，次のように書ける．

```
val ex1' : unit ->
  (all_empty, all_empty, unit data) monad
let ex1' () =
  (* 変数 arr で0番目のスロットを参照する *)
  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
```

[*10] ここで **%lin** は OCaml の構文拡張ポイントを示し，**let%lin** 構文はプリプロセッサによって普通の OCaml の構文木に展開される．

[*11] **#t** は，本来は OCaml の多相ヴァリアントかオブジェクト型 t で定義された型のコンストラクタにマッチするパターンであるが，これを置き換える．

[*12] 'a **larr** は 'a array **lin** の別名であることに注意する (図 1)．

© 2007 Information Processing Society of Japan

```
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 GADT を用いたレンズの構成
Fig. 6 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_empty with
  | _, Data a -> a
let (@>) m l = fun pre ->
  match m (l.get pre) with
  | q, d -> l.put pre q, d
```

Fig. 7 LinMonad の実装
Fig. 7 An implementation of LinMonad

```
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 線形型付き配列操作 API の実装
Fig. 8 An implementation of linearly-typed arrays

```
  let%lin #arr, x = lookup 1 arr in
  dealloc arr
```

例 2.2 の線形型によるプログラムとの構文的な違いは，構文

```
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 レンズパターンのための配列 API
Fig. 9 A linearly-typed array API for lens-patterns

拡張 `%lin` と，レンズパターンの `#` 記号のみである． □

複数の配列操作もまた，レンズパターンにより直観的に記述できる．

Example 4.3 (レンズパターンによる複数の配列操作) レンズパターンを用いると，例 3.4 と同等なプログラムは，次のようになる．

```
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
```

© 2007 Information Processing Society of Japan

```
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 %lin 拡張構文で用いる関数
Fig. 10 Functions for the %lin syntax extension

```
  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 レンズパターンの展開

レンズパターンの意味を，%lin の展開によって与える．まず，%lin のパターンがレンズパターンのみで構成される場合を考え，構造化データの場合に一般化する．

レンズパターンを用いた構文

let%lin #l = e_1 in e_2

は，

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

の略記である．ここで >>- は，バインド関数の一種であり，右辺の関数として fun%lin で書かれた関数を要求する．

関数 fun%lin #l -> e_2 は，構文拡張によって内部的に Bind__(fun tmp -> _put l tmp >> e_2) と展開される．ここで tmp はフレッシュな変数である．Bind__ は，関数が fun%lin によって作られたことを型の上で区別するためのコンストラクタであり，プログラマが用いてはならない．これにより，>>- の右辺はレンズパターンを持つ %lin を用いた関数のみを取ることを静的に保証する．_put l v は値 v をレンズ l が示すスロットに格納するコマンドである．構文拡張 fun%lin は一般に次の型をもつ：

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

ここで α, pre, $post$, β はレンズと e_2 の型によって決まる．例えばレンズ _0 を用いる次のような型付けが考えられる．

```
(fun%lin #_0 -> return ()) :
  ('a lin ->
    (empty * 'pre, 'a lin * 'pre, unit data)
      monad) bind
```

この関数は，線形値を引数に取り，それを空の 0 番目のスロットに格納するコマンドを返す．

構文拡張は，全体として e_1 >>- Bind__(fun tmp -> _put l tmp >> e_2) という内部表現に展開される．変数 tmp は e_1 の計算結果である線形値を束縛後すぐにスロット l に割り当てられ，またプログラム中の他の部分に現れないため，線型性が維持される．

図 10 に %lin 構文拡張で用いる関数のシグネチャとその実装を示す．>>- は通常のバインド操作に加えて，右辺にコンストラクタ Bind__ でラップされた関数をとる．_put は入力のスロット列 pre に対して，レンズで線形値をストアしたスロット列を出力する．

パターンにおける線形値と非制限値の区別 lookup の結果の値の束縛 #arr, x (例 4.2) のような線形値と非制限値が混在したデータを扱うため，変数パターンを導入する．ここで注意しなければならないのは，それ自体は多相性をもつ変数パターンで線形型の値を束縛してはならないことである．ここで，fun%linのパターンのトップレベルは線形値であり，その内部に出現する非制限値は Data でラップされるという前提をおく．さらに，非線形値の内部には線形値が出現しないと仮定してよい．これらの前提のもとで，変数パターンをトップレベルでは用いずに必ず Data でラップし，Dataでラップしない変数パターンは（線形値を束縛する可能性があるため）構文エラーとする制限を加える．このことにより，変数が線形値を束縛しないことを保証する．

パターンの展開規則 変数パターンを常に Data でラップするのは煩雑であるため，パターンの展開は次の規則を用いる．

- Data x のように，明示的なコンストラクタ Data の内側では，変数パターンを許す．
- Data の内側にない変数パターンの型は τ data として，暗黙に Data でラップする．

これにより，例えば lookup の計算結果 'a larr * 'a data を束縛するパターン #arr, x のように，において明示的な Data のコンストラクタが不要になる一方で，型 'a larr * 'a option data のような非線形値に対する構造的なパターンマッチは，#arr, Data (Some x) のように Data を明示することで書けるようになる．

fun%lin に加えて，パターン束縛を伴う構文 function, match をそれぞれレンズパターンで拡張した構文 function%lin, match%lin を導入する．それぞれの展開は次のとおり定める．

- fun%lin pat -> e は，Bind__ (fun $conv(pat)$ -> $puts(pat)$ >> e) のように展開する．ここで $conv(pat)$ は，後に与えるパターン pat を変換する構文の上の関数である．$puts(pat)$ は，pat に出現するレンズ $l_1, \ldots, l_n$，と，それに対応して $conv(pat)$ が生成したフレッシュな変数 $tmp_1, \ldots, tmp_n$ に対して，操作 _put l_1 tmp_1 >> $\cdots$ >> _put l_n tmp_n を生成する関数である．
- function%lin $case_1$ | $\cdots$ | $case_n$ は，pat_i -> e_i の形をもつ各 $case_i$ について，fun%lin と同様に展開する．
- let%lin pat = e_1 in e_2 は，e_1 >>- fun%lin pat e_2 と同様に展開する．
- match%lin e with $case_1$ | $\cdots$ | $case_n$ は，e >>- function%lin $case_1$ | $\cdots$ | $case_n$ と同様に展開する．

%lin 構文におけるパターン束縛の展開アルゴリズム conv を図 11 の擬似コードで示す．関数 _traverse は，パターン p を再帰的に変換する関数である．レンズパターンの場合は，

© 2007 Information Processing Society of Japan

```
let rec _traverse p =
  match p with
  | レンズパターン #l である
    -> fresh な変数 tmp を生成;
       レンズ l と tmp のペアを記録;
       tmp
  | 変数パターン var である
    -> Data var
  | 引数なしコンストラクタパターン C である
    -> C
  | 引数があるコンストラクタパターン
    C(p1,p2,...) である
    -> C(_traverse p1, _traverse p2, ...)
  | _ -> エラー (as パターンなど)
let conv (p : pattern) : pattern =
  if p がレンズパターンである then
    _traverse p
  else
    Lin__ (_traverse p)
    (*線形型のコンストラクタパターンで包む*)
```

Fig. 11 %lin パターンの展開アルゴリズム conv
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 線形型が付いたリスト
Fig. 12 A linearly-typed list

フレッシュな変数に置き換え，その変数とレンズの対応を記録し，後の _put の挿入において用いる．一方，通常の変数束縛の場合は，'a **data** 型のパターン Data x として展開する．コンストラクタパターンについては，引数のパターンについて再帰的に変換する．**as** パターンなど，線形型の値が漏洩する可能性があるパターンにはエラーを返す．

ユーザープログラムの安全性 まとめると，このライブラリと付属する構文拡張を使い，構成子 Lin__ および Bind__ を直接に使わなければ[*13]，**lin** 型の値が線形的に使われていることが保証される．具体的には，線形的な値は常にモナド内のスロット列に保管されており，**fun%lin** 構文で **lin**型の引数を取る関数が作れるものの，その関数にはモナド経由でスロット列から引き出した値しか渡せず，その関数内でもすぐまたスロット列に置かれる．

4.3 線形型付きリスト

線形型付きの構造化データに対するパターンマッチを活用した例として，線形型が付いたリストを図 12 のとおり導入する．さらに，この例題を通して，レンズが指すスロットを直接操作する関数 get_lin, put_lin, put_linval と，限定的な構文で線形型のデータを構築する**線形値コンストラクタ**を導入する．

Example 4.4 (リストの反復による消費) 次の関数 iter0 f は，スロット _0 に割り当てられた線形リストの各要素に f を適用し，リスト全体を消費する．ここで get_lin l はレンズ l が参照するスロットから線形型の値を取り出し，返すとともに，そのスロットを空にする．

```
val iter0 : ('a -> 'b) ->
  ('a linlist * 'xs, empty * 'xs, unit data)
    monad
```

[*13] **lin** と bind が抽象型であるべきなのに，構文拡張で生成されるコードの中で構成子を使うため，実装が隠せないという事情から来る制限である．

```
let rec iter0 f =
  match%lin get_lin _0 with
  | Cons(x, #_0) -> f x; iter0 f
  | Nil -> return ()
```

□

iter0 は簡潔である一方，レンズが_0 に固定されているという問題がある．しかしながら，iter f s のようにレンズを単一の引数で取るのはうまくいかない．

Example 4.5 (単相性による型エラー) 次の関数 iter f s は，型付けできない．

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

これは，get_lin でレンズの型が ('a linlist, empty, 'pre, 'post) lens に固定され，レンズパターンが要求する空のスロットへの割り当てのレンズ (empty, 'a linlist, 'post, 'pre) lens としては用いることができなくなるためである．次のように，使用法が異なるレンズを複数渡すとうまくいく．

```
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 ()
```

この iter' には，iter' f _0 _0 のように同じ多相的なレンズを渡すこともできる． □

線形値コンストラクタ リストの map 関数において，線形値を構築する手段が必要になる．このための構文 [**%linret** c] を次のとおり導入する．

- [**%linret** c] は，線形値 c を結果の値とするモナド値である．
- [**%linret** c] において，e にはコンストラクタの適用のネスト C(c_1, ..., c_n) か，**レンズ参照** !! l のみを許す．
- レンズ参照 !! l は，レンズ l が参照する空でないスロットから取り出した値を返す．
- [**%linret** c] は，それぞれのレンズ参照の出現 !! $l_1, \ldots,$!! l_n が参照するスロットを空にする．

これにより，リストの map が構成できる．

Example 4.6 (線形型が付いたリストの map**)** 次の関数 map0 f は，スロット _0 に割り当てられた線形型のリストを消費し，各要素に f を適用した新しい線形型のリストを _0 に割り当てる．

```
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
```

put_lin l m は，m を実行し，結果の値をレンズ l が参照するスロットに格納する．put_linval l v は，値 v をレンズ l

© 2007 Information Processing Society of Japan

```
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 スロットを直接操作する API
Fig. 13 APIs for direct slot access

が参照するスロットに格納する. □

スロットを直接操作する API のシグネチャ get_lin, put_lin, put_linval のシグネチャおよび実装を図 13 に示す.

Example 4.7 (末尾再帰する rev_map**)** OCaml 標準ライブラリの List.rev_map は，呼び出しスタックを消費しない末尾再帰版の map 関数である．これを真似て，次の関数 rev_map f は，スロット _0 に割り当てられた線形型のリストを消費し，要素を逆順に f を適用した新しい線形型のリストを _1 に割り当てる.

```
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 (map **の一般化 (1))** 次の関数 map' は，例 4.6 の関数 map0 をレンズを引数に取るよう一般化したものである.

```
val map' : ('a -> 'a) ->
  ('a linlist, empty, '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
```

残念ながらこの map 関数は型を変えることができない．古いリストと新しいリストで，リストをスロットから取り出す操作を s1，リストをスロットに格納する操作を s2 で共用しているためである. □

Example 4.9 (map **の一般化 (2))** 古いリストと新しいリストで異なるレンズを用いることにより，一般的な map 関数を構成できる.

```
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. セッション型のエンコード

線形型を用いたプログラミングのより実用的な例として，セッション型の実現と，構造化された線形値のパターンマッチを活用した**サンタクロース問題**の解法を与える.

5.1 セッション型

セッション型 [9] はプログラムにおける通信プロトコルを表現する型であり，型の整合性によって，通信が正しく実行されて正常に終了することを保証する．セッション型は線形型と同様に，セッションの使用回数を追跡する線形性をもつ.

Example 5.1 (セッション型による足し算サービス) セッション型による通信は，チャネルに対してセッションを確立して行う．次のプログラムは，セッション型を用いた足し算サービスの例である：

```
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
```

このプログラムは，スロット _0 において割り当てられたセッションにおいて二つの整数を受信し，その和を送信し，終了する．このような通信が静的に型として現れるのがセッション型の特徴である．ここで用いるセッション型は，型 τ の値を受信して θ へと継続する (θ, τ) recvと，型 τ の値を送信して θ へと継続する (θ, τ) sendと，セッションの終了を表す closeである．足し算サービスは，スロット _0 において

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

の型をもつ[*14]. □

図 14 に，本節で用いるセッション型による通信 API のシグネチャを与える．ここで型 (θ_1, θ_2) channel は，セッションの開始点であるチャネルの型である．チャネルにおいては，accept により通信相手を待ち受けることができ，相手が request による接続要求することで，セッションが確立する．θ_1 は accept する側 (サーバー) が従うセッションであり，θ_2 は request する側 (クライアント) が従うセッションであ

*14 この型表現は，OCaml の後置記法のため，右端から順に通信のステップを表現している

© 2007 Information Processing Society of Japan

```
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 セッション型による通信 API
Fig. 14 Session-typed communication API

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

Fig. 15 双対性を保証するチャネル生成 API
Fig. 15 The channel creation API for ensuring duality

る．セッション型理論において，θ_1 と θ_2 が双対 (dual) と呼ばれる関係にあれば，そのチャネルにおける通信が整合する（デッドロックが起きず，メッセージの型が一致する）ことが保証される．双対性は，チャネルを図 15 に示す API によって生成することで保証する．例えば，`s2c finish` はサーバーからクライアントへの送信の後に終了するセッションをもつチャネル `((close, 'v) send * (close, 'v) recv) channel` を生成する．

5.2 サンタクロース問題

サンタクロース問題 [2], [27] は，Trono によって提案された並行プログラミングの問題の一つであり，様々なプログラミング言語における並行計算機能のベンチマークの一つになっている [18]．サンタクロース問題の概略は次の通りである．

> サンタクロースは眠っており，9 匹のトナカイ (reindeer) すべてが休暇から帰ってくるか，10 人のうち 3 人のエルフ (elf) がやってくると目を覚まして仕事を始める．$\cdots$(略)$\cdots$ もしトナカイとエルフの両方が待っている場合，トナカイに優先権を与える．

モデル化 サンタクロース問題を，次のとおりモデル化する．1 人のサンタがメインスレッドで `accept` を用いてチャネルで待ち受ける．トナカイとエルフには，一匹ずつスレッドを割り当て，ランダムなタイミングで `requst` を用いてサンタとのセッションを確立する．サンタクロースは，トナカイと確立したセッションが 9 つになると，全てのトナカイとのセッション通信を行う．同様に，エルフとのセッションが 3 つになると，エルフとのセッションを処理する．

セッションのリスト トナカイやエルフとのセッションは，サンタが保持する線形のリストに格納する．これにより，動的に増減するセッションを，線形性を損なわずに維持できる．セッションのリストは，`iter`関数により一括して処理する．

```
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 セッションのリスト処理
Fig. 16 Iteration on a list of sessions

セッションのリスト処理の定義を図 16 に示す．型 θ `slist` はセッションのリストである．図 12 の線形な型のついたリストと同様の構造をもつが，型 θ **`lin`** の線形性をもつセッションを保持する点で異なる．関数 `iter` は，`iter i l l l f` の形でのみ用いる．これは，レンズ `l` が参照するリストのうち，`i`個の要素をそれぞれスロット`_0` に格納し，`f` に渡す高階関数である．例 4.5, 4.8, 4.9 にも見られたように，単相性の制限のもとで，同じレンズを異なる型で参照するため，`l1`，`l2`，`l3` の 3 つの仮引数をもつ．

図 17 に，サンタクロース問題の解法のうちサンタクロースの記述を示す（エルフとトナカイの部分についてはより直接的であり平易である）．まず，型 `kind` はエルフとトナカイがそれぞれ自分自身がどちらであるのかを知らせるためのデータ型である．レンズ`_0` は，トナカイやエルフとのセッションの一時的な格納に用いるほか，`iter` による作業領域としても用いる．レンズ`_1`，`_2` は，それぞれエルフとのセッションのリスト，トナカイとのセッションのリストのために用いるため，頭文字をとって `e`, `r` という別名をつける．

まず，サンタは `e`, `r` に空リストを割り当て，ループ `loop` に入る．ループでは，チャネルにおいてセッションを `accept` で待ち受け，確立されたら相手の種別を `receive` により受け取る．もし相手がエルフであれば， `e` に残りのセッションを格納し，エルフのカウント数 `ecount` を増やす．トナカイの場合も同様に， `r` に残りのセッションを格納し，トナカイのカウント数 `rcount` を増やす．それぞれのカウント数が，処理すべき数に達した場合，トナカイには `"Let's deliver!"` という文字列を，エルフには `"Make a new toy!"` という文字列を送り，セッションを終了する．

6. 関連研究

パラメタ化モナドにおいて線形型を扱う理論的な枠組みは Atkey [1] で導入されている．Garrigue が OCaml メーリングリストへのポストで提案した Safeio [7] は，パラメタ化モナドによる，OCaml で利用可能な線型性をもつリソースの最初のエンコーディングであり，本論文の § 3 に大部分が反映されている．

© 2007 Information Processing Society of Japan

```
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 サンタクロース問題のセッションリストによる解法
Fig. 17 A solution for the Santa Claus problem using lists of sessions

6.1 Haskell における線形型の実現

De Bruijn index を用いた埋め込み Kiselyov [14] による finally tagless interprefer は，Haskell において本手法のように型付きのプログラミング言語を埋め込むための技法であり，de Bruijn index を用いた λ 計算の埋め込みが可能である．さらに，その拡張として線形 λ 計算の埋め込みを紹介している．例えば，`int`型の和を計算する関数 $\lambda x.\lambda y.x+y$ は `lam (lam (add (s z) z))` となる．しかし，de Bruiijn index は同一の変数であっても λ 抽象のネストに応じて異なるインデックスをとるため，プログラマにとっては束縛関係を把握しづらい．静的検査の基本的なテクニックは，パラメタ化モナドを用いる点で本手法と類似している一方で，λ 抽象を記述可能であるため，前条件と後条件が表す型文脈が変数のネストに従って大きくなる．Kiselyov の手法は，λ 抽象の実現において Haskell の型クラスを用いている．

HOAS による埋め込み Polakow [24] は，Haskell の型クラスとパラメタ間依存を用いて，線形型をエンコードした．この方法は HOAS [20] を用いるため，本論文の手法のように変数をスロットで管理する必要がなく，de Bruijn index のような可読性の問題もない，より直接的な埋め込みになっている．テクニックは Kiselyov の方法と類似しているが，前条件と後条件において，変数の型ではなく，変数の使用フラグの文脈のみを持って回る点で異なる．Paykin と Zdancewic [19] は，Polakow の手法を，Benton の線形論理の linear/non-linear 表現に基づく，より柔軟な体系 [3] に拡張し，豊富な例を導入している．

Haskell におけるいずれの手法も，変数の使用フラグを型レベルに反映させる際に Haskell の型クラスやパラメタ間依存を用いており，他のプログラミング言語において模倣するのは難しい．また，これらの文献は構造的なデータに対するパターンマッチの方法も導入していない．

6.2 セッション型における線型性の実現

既存のプログラミング言語におけるセッション型のエンコーディングとしては，より古く Neubauer と Thiemann による， Haskell の パラメタ間依存を利用した [17] が，著者らが知る限りでは最も古い．しかしながら，線形なチャネルを同時に一つしか操作できず，一般化しづらい．Pucella と Tov [25] は，パラメタ化モナドを用いて複数のチャネルを扱えるセッション型の枠組みを導入した．この枠組みでは，前条件や後条件を線形なリソースのスタックとみなし，スタックのトップに対するセッション通信プリミティブと，`dig` や `swap` といったスタック操作のプリミティブを用いる．この方法は，Haskell 以外のプログラミング言語においても応用可能である利点がある．しかしながら，スタック操作は見通しが悪く，プログラムは一般に読みづらくなる．これを解決したのは Imai ら [12] の Haskell におけるエンコーディングで HOAS を用いた方法である．Polakow らの方法と同様に，HOAS はホスト言語の変数による直接指定が可能であるため見通しがよい．しかしながら，Polakow らの方法と同様に，エンコーディングにおいて Haskell の型クラスとパラメタ間依存を用いており他のプログラミング言語への応用が困難である．

6.3 表現能力の比較

`linocaml` は，線形 λ 計算の λ 抽象の代わりに，ホスト言語の λ 抽象を用いてスロット列とレンズを経由することにより，線形な値を操作する方法を採用している．

興味深い疑問は，線形 λ 計算の λ 抽象に相当する機能を用いなくても，Kiselyov や Polakow の方法のように線形 λ 計算と同等の表現能力を持つかということである．

まず，線形値をとる引数が一階の値である場合を考える．線形 λ 計算における，$(\lambda x.\lambda y.x+y)$ 42 21 のような λ 抽象や関数適用は，`linocaml` においては，足し算を計算する `add` の型が次のようであるとして，

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

レンズを経由してパラメータを渡すことで記述できる．

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

一方，本論文では，$\lambda f.\lambda x.fx$ のように高階関数を扱う方法については議論しなかった．`fun%lin` は `fun%lin #`l_1 `-> fun %lin #`l_2 `->` のようにはネストできず，線形値を変数ではなくスロットに格納するため，このような関数をエンコードする方法は自明ではない[15]．

[15] 例えば § 3.2.3 の extend, shrink を用いればスロット列の先頭に引数を扱うためのローカルな環境を追加できるものの，既存のスロットの位置が深くなるため Kiseoyov の方法と同様に de Bruijn

© 2007 Information Processing Society of Japan

しかしながら，ホスト言語の関数抽象を用いれば，§3.2.2 (例 3.3, 例 3.4) における iteriM, mapiM のような，スロット操作を伴う高階関数を扱える．さらに，例 3.3 は iteriM に渡した高階関数の内部において次のようにスロット _1 を **更新** (update) している：

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

線形 λ 計算において線形値を内包したクロージャは線形性を持つが，Linocaml におけるクロージャ (fun i x -> ..) や (fun%lin #_0 -> ..) は，線形性の制限がない関数であり，自由に呼び出すことができる．このように表現能力の優劣は自明ではないものの，著者らは同等の表現能力を持つと予想している．

7. おわりに

本論文は，パラメタ化モナドとレンズを用い，OCaml における線形型のエンコーディングを与えた．レンズを用いた線形リソースのハンドルの表現は，Standard ML や Haskell といった他のプログラミング言語でも利用可能である利点がある．更に，OCaml の構文拡張機構を用いてパターンマッチ構文を拡張し，線形型をもつリスト等の構造化されたデータの操作方法を与えた．セッション型と線形型のパターンマッチを活用し，サンタクロース問題を例として，より応用に近い通信プログラミングにおける有用性を示した．このエンコーディングは，linocaml ライブラリとして利用可能である．

linocaml の長所は，モナドやレンズといった関数型プログラミングにおける抽象を組み合わせて，軽量な形で線形型を用いたプログラミングを模倣可能なことである．線形型を用いたプログラミングは，Rust を除けば実用規模のプログラミング言語にはこれまで導入されていない．幅広い応用範囲をもつ OCaml に線形型導入したことにより，リソースの安全性や効率性の観点で様々な恩恵が得られるようになることが期待できる．

今後の課題 以下に今後の課題を述べる．

LinMonadを用いた計算はクロージャの生成を多用するため，ダイレクトスタイルと比較して効率が悪化することが予想される．また，レンズを用いたスロットのアクセスは，コンスを辿るためにわずかなコストがかかる．このコストは通信プログラムのように他の要因で遅延がある状況では無視できるほど相対的に小さいと考えられるが，配列操作のように高い効率が要求される場面では望ましくないかもしれない．モナドとレンズを用いた場合の計算効率の悪化と改善は，今後の課題である．

レンズは_0 : ('a, 'b, 'a * 'xs, 'b * 'xs) lens のように多相的な型を持つ．このような第 1 級の多相的な値は多くのプログラミング言語では利用できない．Java 等のプログラミング言語でレンズを用いたプログラミングを実現し，より広い応用範囲を得るのは今後の課題である．

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.

index に起因する読みづらさの問題が起こると考えられる．

References

[1] Atkey, R.: Parameterized Notions of Computation, *Journal of Functional Programming*, Vol. 13, No. 3-4, pp. 355–376 (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 (Extended Abstract), *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, pp. 121–135 (online), DOI: 10.1007/BFb0022251 (1994).

[4] Bernardy, J., Boespflug, M., Newton, R. R., Peyton Jones, S. 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] (editor), S. M.: Haskell 2010 Language Report (2010). https://www.haskell.org/definition/.

[6] 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).

[7] Garrigue, J.: Safeio (A mailing-list post) (2006). Available at https://github.com/garrigue/safeio.

[8] Garrigue, J. and Normand, J. L.: Adding GADTs to OCaml: the direct approach (2011). In *ACM SIGPLAN Workshop on ML 2011*. Available at https://www.math.nagoya-u.ac.jp/~garrigue/papers/ml2011.pdf.

[9] Honda, K., Vasconcelos, V. T. and Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming, *ESOP '98: Proceedings of the 7th European Symposium on Programming*, LNCS, Vol. 1381, Springer, pp. 122–138 (1998).

[10] 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, Springer, pp. 99–118 (online), DOI: 10.1007/978-3-319-59746-1_6 (2017).

[11] 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.

[12] Imai, K., Yuen, S. and Agusa, K.: Session Type Inference in Haskell, *PLACES 2010: Thrid Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, EPTCS, Vol. 69, pp. 74–91 (2010).

[13] Jones, M. P.: Type Classes with Functional Dependencies, *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, Springer, pp. 230–244 (2000).

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

[15] Kmett, E.: Lenses, Folds and Traversals (2012). Available at http://lens.github.io/.

[16] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. and Vouillon, J.: Representation of OCaml data types, in *The OCaml system release 4.07 Documentation and user's manual* (2018). Available at http://caml.inria.fr/pub/docs/manual-ocaml/intfc.html.

[17] Neubauer, M. and Thiemann, P.: An Implementation of Session Types, *PADL'04 : Practical Aspects of Declarative Languages*, LNCS, Vol. 3057, Springer, pp. 56–70 (2004).

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

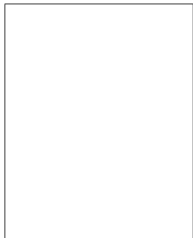
[19] Paykin, J. and Zdancewic, S.: The linearity Monad, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pp. 117–132 (online), DOI: 10.1145/3122955.3122965 (2017).

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

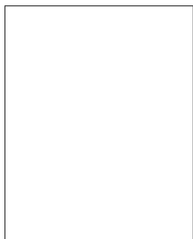
[21] Pickering, M., Gibbons, J. and Wu, N.: Profunctor Optics:

© 2007 Information Processing Society of Japan

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). https://clean.cs.ru.nl/Clean.
[24] Polakow, J.: Embedding a Full Linear Lambda Calculus in Haskell, *Haskell '15: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, ACM, pp. 177–188 (online), DOI: 10.1145/2804302.2804309 (2015).
[25] Pucella, R. and Tov, J. A.: Haskell Session Types with (Almost) No Class, *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, ACM, pp. 25–36 (online), DOI: 10.1145/1411286.1411290 (2008).
[26] Rust project developers: The Rust Programming Language. 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).
[29] Wadler, P.: The essence of functional programming, *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, ACM, pp. 1–14 (online), DOI: 10.1145/143165.143169 (1992).



Keigo Imai received Doctor of Information Science from Nagoya University in 2012. He was at 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 パリ第 7 大学でで修士課程を修了した後，1995 年に東京大学で理学博士を取得．京都大学数理解析研究所助手の後，2004 年より名古屋大学多元数理科学研究科で助教授・准教授・教授．プログラミング言語理論，特に型システムとプログラムの証明に興味を持つ．情報処理学会，日本ソフトウェア科学会，ACM 会員．

© 2007 Information Processing Society of Japan