情報処理学会論文誌　プログラミング　Vol. 2　No. 5　1–14 (Nov. 2009)

スクリプト言語コンパイラのための評価による最適化

戸澤 晶彦[†1]　小野寺 民也[†1]

近年，PHP，Java Script，Ruby，Perl などのスクリプト言語の人気が高まっている．これらの言語に特徴的なのは実行系やアプリケーションにおいて連想配列が幅広く使われることである．本稿ではスクリプト言語のコンパイラ向けに連想配列の処理を削減するための評価器による最適化の実現方法を報告する．まずプログラムからコンパイル時（static）と実行時（dynamic）の値の組への評価をするような単純な評価器から始め，これに 1) 連想配列のコンパイル時表現，2) 不要な `let` の除去，3) コンパイル時連想配列の実体化の処理などを追加していくことで実用的な連想配列の最適化が可能になる．

Optimization by Evaluation in Compilers for Scripting Languages

Akihiko Tozawa[†1] and Tamiya Onodeara[†1]

Scripting languages including PHP, JavaScript, Ruby, Perl, etc., are recently very popular. One characteristic of these languages is the heavy use of associative arrays in their runtime and applications. In this talk, we report how, inside compilers for scripting languages, optimization-by-evaluation techniques can optimize associative arrays. Starting from a simple evaluator that evaluates a program into a pair of compile-time (static) and runtime (dynamic) values, we derive a practical optimizer for associative arrays step by step by adding 1) compile-time array representation, 2) removal of useless `let`-insertion, and 3) reification of compile-time arrays.

†1 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Research, Tokyo Research Laboratory

1. はじめに

近年，PHP，Java Script，Ruby，Perl などのスクリプト言語の人気が高まっている．これらの言語に特徴的なのは実行系やアプリケーションにおいて連想配列が幅広く使われることである．

PHP を例にとると，多くの PHP アプリケーションで連想配列はモジュール間のデータのやりとりで使われる基本的なデータ構造である．また PHP の実行系の内部においても連想配列はローカルおよびグローバル変数テーブル，オブジェクトのフィールドの内部表現，定数テーブル，関数テーブルなど，あらゆる箇所で使われる．著者らは近年コンパイラに基づく PHP 実行系 P9[9] を試作しているが，SugarCRM という大規模なアプリケーションでの実験では約 30%の処理時間が実行時ライブラリ中の連想配列操作に費されていた．

そこで，本稿ではスクリプト言語のコンパイラのために「プログラム中の連想配列操作を可能なかぎり取り除く」という最適化の問題を考える．

いま連想配列の基本操作は `load` 操作と，`store` 操作であるから，一見このためには既存の手法，たとえば命令型言語コンパイラの仮想レジスタ操作レベルの最適化手法が使えそうである．しかし細かくみていくと，たとえば連想配列の走査（PHP の `foreach` 文など）の取扱いなど，仮想レジスタ操作と連想配列操作は異っており，この違いをどう解決するかは容易ではない．

本稿ではこれに代わる手法として，主に関数型言語コミュニティで研究されている部分評価あるいは評価による正規化（NBE = normalization by evaluation）という手法を応用した連想配列操作の最適化器を提案する．この最適化器は OCaml で 300 行程度で実装され，本稿中にすべてソースコードがのるほど小さな手間で実現できるにもかかわらず，かなり強力な最適化が可能である．

論文の構成は次のようである．まず次章でベースとなる評価器を導入し，さらに PHP を例にとって目指す最適化の目標をあげる．3 章では目標に向けて最適化器をどう改良してゆくかについて述べる．4 章は既存研究との比較を議論する．

2. 準　　備

2.1 評価による最適化

部分評価[2),3),7)] あるいは評価による正規化[4),6)] という手法による最適化は考えかたが非常にシンプルであるにもかかわらず，強力な最適化が可能である．「評価」といっても通常

の実行のための評価と異なり，部分評価器や NBE 評価器の出力は実行結果の値とは別の特別な値（symval）である．このような評価器にはいくつか種類があるが，本稿では symval としてコンパイル時の値（static）とコンパイル時に分からない値を表す実行時のコード（dynamic）の組を扱うような評価器を用いる．

図 1 のように評価器 eval を定義する．この関数は symval hexp -> symval * residual という型を持つ．たとえば $(\lambda x.x)(\lambda x.x)$ に対して eval を実行した場合，以下のような結果が得られる．

```
# let (s, d), r = eval (% (fun x -> x) (fun x -> x) %);;
val s : static = SFun <fun>
val d : dynamic = Abs (["_1"], Var "_1")
val r : residual = <fun>
# r d;;
- : exp = Abs (["_1"], Var "_1")
```

% と % で囲まれた部分は symval hexp 型の入力のソースプログラムを見やすくするためのシンタクスシュガーである[⋆1]．eval は入力を値 (s, d) : symval と，その外側にある let 文の列 r : residual の組に評価する．今回は r は空なので，実行時コード d : dynamic そのものが簡単になった出力結果のプログラムとなる．また，コンパイル時の値 s : static はすべて表示されていないが，symval 値を自分自身と空の residual 値の組に評価するような関数に SFun というタグをつけたものになっている．

少し補足説明をする．この評価器はたとえば Sumii ら[7] が定義した副作用のある値呼び言語のためのオンライン部分評価器とほぼ同じものである[⋆2]．評価器の入力は高階抽象構文（'a hexp）で表現されている．これはメタ言語（OCaml）の関数抽象を対象言語の関数抽象のシンタクスの中で使うもので，特にこれでなくてはならないわけではないが，環境を明示的にせずにすんで記述がシンプルになるのでこれを使っている．評価器の出力は通常の抽象構文（exp）になっている．また，図 1 の slet 関数と rlet 関数は，関数適用のところに let-文を入れて副作用の移動や式のコピーを避ける，いわゆる let-insertion[7] あるいは残余モナド[4] の機能を実現している．

評価器 eval は構文に従って再帰的にプログラムを評価して (s, d) : symval を計算し，それと同時に評価中にであった動的な関数適用に関する let-文の列 r : residual を計算してゆく．

- 変数 HVar sv（シンタクスシュガーでは % sv %）は，sv 自身に評価され，空の residual が残る．
- 関数抽象 HAbs(n, fun[$\mathtt{sv}_1$; ...; $\mathtt{sv}_n$]->e)（シンタクスシュガーでは % fun($\mathtt{sv}_1$, ... $\mathtt{sv}_n$)-> { e } %[⋆3]）を評価するとき，結果の static 値は関数値 SFun f であって，その f は symval list 型の引数をもらい，e の中の $\mathtt{sv}_1, \ldots, \mathtt{sv}_n$ の出現をこの引数値に置き換えたプログラムに eval を適用した結果値を返すものである．また dynamic 値は Abs 式でその実体は，この f を静的な値が分からない変数の列であることを表す (SNone, Var $\mathtt{x}_1$), ..., (SNone, Var $\mathtt{x}_n$) なる引数の列に適用し，得られた結果の dynamic 値から求められる．ここでは，f の適用で残される let-文の列を Abs 式の内部に閉じこめるために rlet 関数が使われている．一方変数のケースと同様に関数抽象自体に副作用はないので，外側に残る residual 値は空である．
- 最後に関数適用 HApp(e, [$\mathtt{e}_1$; ...; $\mathtt{e}_n$])（シンタクスシュガーでは % { e }({ $\mathtt{e}_1$ }, ..., { $\mathtt{e}_n$ }) %）を評価するとき，eval e の静的な関数値が知られている場合，これを各引数を評価した結果に適用した結果を返す．このときの residual 値は e, $\mathtt{e}_1, \ldots, \mathtt{e}_n$ を評価したときのそれぞれの residual 値の合成として計算される．もし，eval e の静的な関数値が分からない場合は，適用の結果の静的な値も分からないことを示す (SNone, Var x) という組を返すが，この変数 x を定義するコードは slet 関数による let-insertion によって，結果の residual 値に追加される．

2.2　最初の試み

本稿ではこの単純な評価器から PHP のようなより複雑な言語の最適化器を導出することを考える．より複雑といっても条件文や再帰などは既存の手法で取り扱えばよく，またスカラー値に対する組み込みの操作を定義するのも容易である．いずれも symval 型の関数値（たとえば**図 2** の ifop，eq）を適切に定義すればよい．

しかし，連想配列のようなネスト可能なデータ構造はすこし難しい．単純にスカラー値と同じような取扱いはしたくはない．そのような扱いでは配列の中身がすべてコンパイル時に

⋆1 OCaml では camlp4 を使ってこのようなシンタクスシュガーが容易に定義できる．

⋆2 細かい違いとして，1) 彼らは関数抽象なども let-挿入しているが，我々はしていない点，2) 彼らは状態ベースの let-挿入を提案しているが，我々はこれと同等な処理を蓄積モナド（accumulating monad[4]）の手法で行っている点．3) また，我々は中間言語として通常の λ 計算ではなく，複数パラメータの受け渡しが扱えるような拡張を使っている点などがあるが，本質的には同じである．

⋆3 % ... % 内の中括弧はシンタクスシュガーをエスケープするのに使う．

```
open List

(* 出力コードの抽象構文 *)
type exp =
 | Var of string
 | Abs of string list * exp
 | App of exp * exp list
 | BoolConst of bool
 | IntConst of int
 | StrConst of string
 | NullConst

(* 入力コードの高階抽象構文 *)
type 'a hexp =
 | HVar of 'a
 | HApp of 'a hexp * 'a hexp list
 | HAbs of int * ('a list -> 'a hexp)

(* 評価器の出力の型 *)
type static =
 | SNone
 | SConst
 | SFun of (symval list -> symval * residual)
and dynamic =
 exp
and residual =
 exp -> exp
and symval =
 static * dynamic

(* 識別子の生成 *)
let fresh, init =
 let cnt = ref 0 in
 let fresh () =
  let nid = "_" ^ string_of_int !cnt in
  let _ = cnt := !cnt + 1 in
  nid
 in
 let init () = cnt := 0 in
 fresh, init

(* いくつかの補助定義 *)
let empty cont = cont
let comp f g x = f (g x)
let dyn sv = snd sv
let _Var x = Var x
let _App (d, ds) = App (d, ds)
let _BoolConst b = BoolConst b
let rec create f n =
 if n = 0 then []
 else f () :: create f (n - 1)

(* let-insertion の仕組み *)
let slet r sv =
 if fst sv = SConst then sv, r else
 let x = fresh () in
  (fst sv, _Var x),
  fun cont ->
   r (App (Abs ([x], cont), [dyn sv]))
let rlet (sv, r) = r (dyn sv)

(* 評価器 *)
let rec eval e =
 match e with
 | HVar sv -> sv, empty
 | HAbs (n, h) ->
   let f = comp eval h in
   let s = SFun f in
   let d =
    let xs = create fresh n in
    Abs (xs,
     rlet (f (map (fun x -> SNone, _Var x) xs)))
   in
   (s, d), empty
 | HApp (e, es) ->
   let (s, d), r = eval e in
   let args, r' =
    fold_right
     (fun e (args, r') ->
      let sv, r = eval e in
      sv :: args, comp r r')
     es ([], empty)
   in
   let r = comp r r' in
   (match s with
   | SNone ->
     slet r (SNone, _App (d, map snd args))
   | SFun f ->
     let sv, r' = f args in
     sv, comp r r')

let conv e = let _ = init () in rlet (eval e)
```

図 1　OCaml 上に実装された最適化のためのシンプルな評価器
Fig. 1　Simple optimization-by-evaluation evaluator in OCaml.

分かっている場合の単なる定数畳みこみ以上の効果が得られないからである．たとえば連想配列のキーの部分だけは分かっているが，値の部分は分からないというような部分的な情報も利用して，プログラムを簡単化したい．なるべく評価器や対象言語の構文を変えずにこれを可能にする試みとしては対象言語のシンタクスの上にデータ構造をエンコードしてしまう方法がある．具体的にはいま扱っているのは λ 計算なのだからチャーチ符号化が考えられる．

本稿で考えている中間言語上では以下のような連想配列の操作が定義されているとする．

- `null`：空の連想配列を表す．
- `store`(k, v, a)：連想配列 a にキーと値の組 (k, v) を追加した新しい連想配列を返す．
- `load`(k, a)：連想配列 a のキー k に対応する値を返す．
- `foreach`(c, n, a)：連想配列 a の破壊子 c と n による走査を行う．

最後の `foreach` について詳しくは後述する．いま `foreach` 以外の操作のコンパイル時の

```
(* 組み込みの eq と ifop *)
let eq =
 SFun (fun [sv1; sv2] ->
  match fst sv1, fst sv2 with
  | SConst, SConst ->
    (SConst, _BoolConst (dyn sv1 = dyn sv2)),
    empty
  | _ ->
    eval (% ‘eq (sv1, sv2) %)),
 _Var "eq"

let ifop =
 SFun (fun [sv1; sv2; sv3] ->
  match fst sv1 with
  | SConst when dyn sv1 = BoolConst true ->
    eval (% sv2 () %)
  | SConst when dyn sv1 = BoolConst false ->
    eval (% sv3 () %)
  | _ ->
    eval (% ‘ifop (sv1, sv2, sv3) %)),
 _Var "ifop"
```

図 2　ifop と eq の定義
Fig. 2　Definitions of ifop and eq.

（symval 値としての）意味をチャーチ符号表現の操作として以下のように定義してみよう．

```
let null =
   fst (eval (% fun (c, n) -> n %))
let store =
   fst (eval (% fun (k, v, a) -> fun (c, n) -> c (k, v, a (c, n)) %))
let load =
   fst (eval (% fun (k, a) ->
                a (fun (k', v, x) -> fun () -> if eq (k, k') then v else x (),
                   fun () -> ())
                () %))
```

ここで考えているのは**図 3** のようなキーと値の組のリストのチャーチ符号化である．ただし図 3 は通常のリストの $[(\mathtt{k0},\mathtt{v0});(\mathtt{k1},\mathtt{v1});\ldots]$ に対応する．

いま fst(eval...) の部分は定義している関数の symval 値を取り出している．定義の実体は null については空リストのチャーチ符号化であり，また store については，リスト a の先頭に k と v の組を追加する処理である．

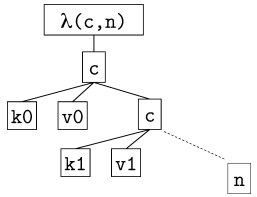

図 3　連想配列のチャーチ符号表現
Fig. 3　Church encoding of associative array.

load 処理[⋆1]は，1) リストのチャーチ符号化 a が通常のリストの fold_right 関数に対応すること，2) その蓄積パラメータ x は遅延 fun () -> ... の形で計算されていることが分かると理解しやすい．すなわち，

- 1) についてより詳しく述べると，$\mathtt{a}(c,n)$ という適用はチャーチ符号化リストの中の c と n を実際のリストの破壊の処理 (c,n) に置きかえることに対応する．たとえばリスト $[(\mathtt{k0},\mathtt{v0})]$ に対応する $\mathtt{c}(\mathtt{k0},\mathtt{v0},\mathtt{n})$ について load の定義中の $\mathtt{fun}(\mathtt{k'},\mathtt{v},\mathtt{x})\ \mathtt{->}\cdots$ と $\mathtt{fun}\,()\ \mathtt{->}\cdots$ をそれぞれ c と n へ代入して簡約すると fun () -> if eq(k, k0) then v0 else () という期待されるロードの処理になる．このようなチャーチ符号化リストの破壊の手法は本稿中で繰り返し使う．
- 2) の遅延は不要な eq および ifop の適用が残余コードに残らないようにするためである．

これらの定義を追加してから，評価器を実行すると以下のような結果が得られる．ただし %...% に現れる ‘v, ‘echo などは自由変数を表現していて HVar (SNone, Var "v") などを表すものとする．また let x = e1 in e2 は (fun x -> e2) e1 の省略である．print は結果コード（exp 値）を見やすく表示している．

```
# print (conv (% let a = store ("x", ‘v, null) in
                 ‘echo (load ("x", a)) %));;
let _19 = echo v in _19
```

⋆1 この定義に出現する if eq (k, k') then v else x () は，ifop (eq (k, k'), fun () -> v, fun () -> x ()) の省略である．

```
- : unit = ()
# print (conv (% let a = store ("x", ‘v, null) in a %));;
fun (_2, _1) -> let _3 = _2 ("x", v, _1) in _3
- : unit = ()
```

最初の結果は期待どおりであり，load("x",a) の処理が結果コードから除去され，さらに連想配列 a が結果コード中ですでに使われないことから不要になった store も除去されている．連想配列 a には変数 v のような実行時コードが含まれるので単なる定数畳み込みでは，このような結果は得られないことに注意する．一方，2 番目のプログラムのような配列自体が式の結果に残るケースでは，配列のかわりにそのチャーチ符号化が結果コードに残ってしまう．このような結果はあまり望ましくない．なぜなら，実行時の連想配列はチャーチ符号やリストではなく，より効率の良い平衡木やハッシュテーブルのようなデータ構造であってほしいからである．

2.3 目　　標

通常の実用言語でデータ構造操作の最適化を考えるときにはエイリアスの問題をまず解かなくてはならないが，これは今回の我々の興味ではない．本稿で取り扱う言語では連想配列はエイリアスを含まないものと仮定する．さらに連想配列の複製はポインタのコピーではなく値の深いコピーであると考える．たとえば PHP の連想配列や OCaml などの有限マップなどは，意味的にはこのようなコピーによる複製を行っていると考えてよい．より正確にいえば PHP の場合，=& 文などによる参照生成がない範囲ではエイリアスは現れない[8]．

このようなエイリアスの制限をおいたとしても，簡単には解けない問題がいくつかある．問題の実例として以下ではいくつか PHP プログラムをあげる．PHP のような命令型言語のプログラムを SSA 変換などにより本稿の中間言語（’a hexp の式）に変換する手法はよく知られているので詳述しないが，そのかわり以下の例では左に最適化したい PHP プログラムを，右に本稿の中間言語での表現をあげる．

解決済みロード操作の除去．

```
$a = null;                          let a = null in
$a["x"] = $x;                       let a = store ("x", ‘x, a) in
$a["y"] = $y;                       let a = store ("y", ‘y, a) in
echo $a["x"];                       ‘echo (load ("x", a))
```

この PHP プログラムのような例では echo $a["x"] でのロード操作を除去して echo $x のような処理に置き換えたい．このような置き換えは前節でみた手法でできているので解決済みとする．

目標 1 ストア操作を残余コードに残す．

```
function f($x, $y) {                fun (x, y) ->
  $a = null;                           let a = null in
  $a["x"] = $x;                        let a = store ("x", x, a) in
  $a["y"] = $y;                        let a = store ("y", y, a) in
  return $a;                           a
}
```

前節で問題になったところである．連想配列 $a が継続のプログラム，つまり現在のコンパイル単位の外で使われるケースで，この場合はチャーチ符号表現などではなく，実行時の連想配列を継続プログラムに渡すようにしないと性能が落ちるだろう．かといって実行時の操作の定義を使って，部分評価を行うというのはうまくいかない[*1]．むしろこのケースの store は展開するのではなく，そのまま残余プログラムに残すべきだと考えられる．

目標 2 連想配列の最初の形が分からないケース．

```
function f($a, $x, $y) {            fun (a, x, y) ->
  $a["x"] = $x;                        let a = store ("x", x, a) in
  $a["y"] = $y;                        let a = store ("y", y, a) in
  echo $a["x"];                        ‘echo (load ("x", a))
}
```

上のケースで $a の値がコンパイル時に不明だったとしてもエイリアスがないという仮定のもとなら echo $a["x"] を echo $x に書きかえるのは安全である．このケースも前節の手法ではうまく行かない．

目標 3 配列の走査とストアの重複の問題．

```
$a = null;                          let a = null in
$a["x"] = $x;                       let a = store ("x", ‘x, a) in
$a["y"] = $y;                       let a = store ("y", ‘y, a) in
$a["x"] = $z;                       let a = store ("x", ‘z, a) in
foreach ($a as $k => $v) { echo $v; }   foreach (fun(k, v, x) -> ‘echo v, (), a)
```

*1 実行時のストア操作というのは実際にはハッシュテーブルや平衡木の操作であり，その定義をそのまま使って部分評価を行うことは（特にキーに動的な部分がある連想配列については）ほぼ不可能だろう．

前節の手法のようにストア操作を単に組のリストへの追加として扱うのは，ロード操作が直近の更新だけを見るならば安全である．しかし多くの言語は連想配列の走査の機能を備えている．この場合には，同じキーについての重複したストアを取りのぞいて走査しなければならない．具体的には上のプログラムでは，最初の `$x` のストアは消え，実行結果は `echo $z; echo $y;` と等価になるはずである．

目標 4 ストアの重複の除去．

また，上の例で `$a` 自体を `return $a` のように返すならば目標 1 のケースと同様に `$a` を作るためのストア操作を残余コードに残す必要がある．ただし 1 回目のストア $\texttt{\$a}[\texttt{"x"}] = \texttt{\$x}$ は上書きされているので，これは除去することができるはずである．

なぜ上であげたような最適化が有用なのかについて補足する．**解決済み**のケースのようなロード操作の除去は非常に有用である．たとえば PHP の実アプリケーション中には関数呼び出しの引数に名前をつけるために，連想配列 `$a` を作成してから `f($a)` のように渡すという用法が多くみられる．このような呼び出し `f($a)` をインライン展開した場合，上のような形のプログラムになる．

目標 3 はやはりインライン展開後のコードで，ロードの代わりに `foreach` が連想配列の破壊子となるケースをカバーしている．**目標 4** のようなストアの重複の除去は，たとえばループ中で繰返し配列を同一キーに対して更新するようなコードでループを部分的に展開したケースで使える．**目標 2** は連想配列の初期値を与えるコードが最適化器から見えないが，更新している部分は見えているという状況でも操作の除去を可能にする．関数などの小さな単位でモジュラーに動作するコンパイラ最適化器として，このようなケースも対応できた方がよい．

最後に**目標 1** は書き換えに基づく通常の最適化ならばストアを書き換えないだけでよいところなので，プログラムをすべて作り直す評価に基づく最適化特有の課題である．ただし書き換えに基づくたとえば命令型コンパイラの最適化手法にも限界があり，これは最後に論ずる．

3. 最適化器の改良

3.1 コンパイル時連想配列の導入

2 章でみたように連想配列を関数として表現し，そのコンパイル時の値をもとからあった `SFun` の値にエンコードする方法には限界があった．問題を解決するためには新しく `SAssoc` 項を `static` 型の宣言に追加する必要がある．

```
type static = ...
  | SAssoc of assoc
and assoc = {
 ordered : bool;
 dstr : 'a. (symval * symval * symval * 'a ans -> 'a ans) * 'a ans -> 'a ans }
and 'a ans =
 Unknown | Ans of symval * residual | Temp of 'a
and ...
```

この宣言の `dstr` レコードメンバの型は多相的に定義されている．OCaml ではレコードメンバのところで一級の型多相が使えるのでこのような定義が可能になる．この型は 2.2 節でみたようなチャーチ符号表現の型ではあるが，

1) 対象言語（`symval hexp`）の上ではなく，メタ言語である OCaml 上でチャーチ符号表現を作る．
2) 2.2 節のようなキーと値の組のリストの型と同型の型 $(\texttt{symval} \times \texttt{symval} \times \alpha \to \alpha) \to \alpha \to \alpha$ ではなく，キーと値に加えて後続リストの `symval` 値自体の 3 つ組のリストのような型と同型な型 $(\texttt{symval} \times \texttt{symval} \times \texttt{symval} \times \alpha \to \alpha) \to \alpha \to \alpha$ になっており，さらに α のところが `'a ans` になっている．

という違いがある．1) については，前章で述べたようにチャーチ符号表現自体を残余プログラムに残す意味がないので，効率の面から対象言語上の構文的な記述は必要ないためである[*1]．2) の 1 つめの理由は初期値が不明であるような連想配列に対応するためである．また `'a ans` 型の各タグの用法はこれから見ていく．`assoc` 型の `ordered` メンバについては次節で説明する．

このような `assoc` 型の値を作る操作は次のように定義できる．

```
let nil = { ordered = true; dstr = (fun (c, n) -> n) }
let cons k v a =
  match fst a with
  | SAssoc m ->
    { ordered = false; dstr = (fun (c, n) -> c (k, v, a, m.dstr (c, n))) }
```

*1 いま `dstr` メンバをチャーチ符号化リストとせず通常の OCaml のリストを使うこともできる．しかし，実行の効率上はチャーチ符号化リストと通常のリストでほぼ同じはずであり，また後述する短絡融合による最適化はチャーチ符号の方が適している．

```
  | _ ->
    { ordered = false; dstr = (fun (c, n) -> c (k, v, a, Unknown)) }
```

値 nil は 1 点の，cons 操作は組（k, v, a, ans）のチャーチ符号表現を作っている．2.2 節のチャーチ符号表現との違いは破壊子 c に後続リスト a : symval，および a に対する破壊子 c，n による蓄積パラメータの計算結果 ans : 'a ans を作って両方を渡すという点である．もし a が動的，すなわちその SAssoc 値が不明ならばこの ans は Unknown になる．

この定義の上で store と load 操作の symval 値を**図 4** のように定義する．新しい定義のもとでの実行結果の例をあげる．

```
let null = SAssoc nil, _Var "null"

let store =
 SFun (fun [k; v; a] ->
    slet empty (SAssoc (cons k v a), _App (_Var "store", [snd k; snd v; snd a]))),
 _Var "store"

let mkans r e =
 let (sv, r') = eval e in Ans (sv, comp r r')

let load =
 SFun (fun [k; a] ->
  match a with
  | SAssoc m, _ ->
    let c (k', v, a, ans) =
     match ans with
     | Ans (x, r) ->
       mkans r (% fun () -> if eq (k, k') then v else x () %)
     | _ ->
       mkans empty (% fun () -> if eq (k, k') then v else `load (k, a) %)
    and n =
     mkans empty (% fun () -> () %)
    in
    let Ans (x, r) = m.dstr (c, n) in
    let x, r' = eval (% x () %) in
    x, comp r r'
  | _ ->
    eval (% `load (k, a) %)),
 _Var "load"
```

図 4　コンパイル時配列の store と load 操作
Fig. 4　store and load operations for compile-time arrays.

```
# print (conv (% let a = store (1, `x, `z) in
                 let a = store (2, load (1, a), a) in
                 store (3, load (0, a), a) %))
let _11 = store (1, x, z) in
let _17 = store (2, x, _11) in
let _27 = load (0, z) in
let _28 = store (3, _27, _17) in
_28
- : unit = ()
```

ここでは前節の目標 1，2 が達成されていることが分かる．すなわち，1) 式の結果として連想配列のチャーチ符号表現を残すのではなく，残余コード中の store を使って実行時に作られる配列を残していて，また 2) 連想配列の初期値 z は不明であるが，そのケースでも不要な z の関数適用などを残余コードに残さずに load 処理が簡単化できている．

図 4 の定義を説明する．まず store 操作は結果の static 値を cons 操作によって作る．また dynamic 値は残余コードに store(k, v, a) 自体を出力するような処理を行っている．この処理によって，前節の目標 1 を達成することができる．

load の定義では 2.2 節の定義と同様のアイディアを使っている．ただし，破壊子 c と n が計算する symval * residual 型の結果値には Ans タグがつく．また，2.2 節では蓄積パラメータ x : symval は，つねにコンスセルの破壊子 c に渡されていた．今回の定義では，これは

1) c に渡されている ans が Ans タグの値ならば後続リストに対して蓄積パラメータが計算できたということであり，ans が蓄積パラメータ x を保持している．
2) さもなければ，すなわち ans が Unknown ならば，load 操作を後続リストに対して行うような式を dynamic 値とするような値を作ってこれを返す．

というようになっている．上記の例の load(0, z) はこの 2) の仕組みによって出力されている．

3.2　配列の走査

PHP の foreach 文など多くの言語が連想配列の走査のための構文を備えている．本稿では foreach(c, n, a) という記述で配列 a を走査する．ここで n は蓄積パラメータの初期値，c はキーと要素値と蓄積パラメータを受けとって，蓄積パラメータを更新する処理とする．

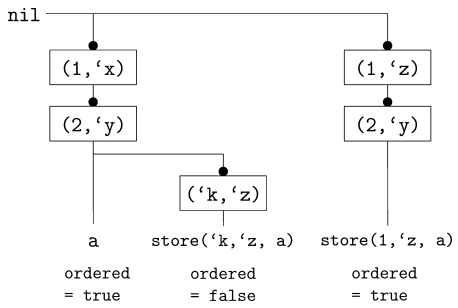

図 5　キーの重複の除去
Fig. 5　Removing key-duplication.

つまり，foreach による走査は関数型言語の fold_right 関数[⋆1]とほとんど同じである．唯一の違いは，連想配列とリストの違いである．連想配列ではキーが重複した場合のストアは上書きになる．前節で定義した store のように先頭への追加のみだと評価器が foreach を展開するときに同一キーの要素を 2 回走査する可能性があって問題になる．

リストからキーの重複を除去するような store を考える．ただし**図 5** に図解するように，すべての重複が除去できるわけではなく，たとえばキー k のコンパイル時の値が不明であれば，ストアが既存のエントリを更新すべきなのか分からない．この場合はエントリはとりあえず先頭に追加するが，同時に assoc 型の ordered メンバの値を false に設定する．一方もしキーの静的な値が分かっているならば，コンパイル時に既存のエントリを更新するか新しいエントリを先頭に追加するという判断をしてリストを更新できる．このとき，元のリストが ordered ならば新しいリストの ordered の値も true となる．

図 6 に新しい store の定義を，**図 7** に foreach の定義を示す．store の定義は図 5 のようなキーの重複の除去を実現している．このような処理のためには連想配列を調べて必要な部分を作りなおす必要があるが，これは m.dstr(c,n) が assoc 型の蓄積パラメータを（正確には assoc ans 型の Temp 値として[⋆2]）計算することで実現している．また cons 処理は ordered である配列に定数キーを追加したケースのみ，結果を ordered とするように定義している．最後に foreach は ordered である連想配列しか壊せないようにしてあ

⋆1 Haskell コミュニティでは foldr 関数．
⋆2 このように ’a ans 型の Temp 項は，リストの破壊操作 m.dstr から多相的にさまざまな型で答えを得るために使っている．

```
let nil = { ordered = true; dstr = (fun (c, n) -> n) }

let cons k v a =
  match fst a with
  | SAssoc m ->
    { ordered = m.ordered && fst k = SConst;
      dstr = (fun (c, n) -> c (k, v, a, m.dstr (c, n))) }
  | _ ->
    { ordered = false; dstr = (fun (c, n) -> c (k, v, a, Unknown)) }

let store =
 SFun (fun [k; v; a] ->
  slet empty
  ((match a with
  | SAssoc m, d ->
    let c (k', v', a, ans) =
     let t = fst (eval (% eq (k, k') %)) in
     match fst t, ans with
     | SConst, _ when dyn t = BoolConst true ->
       Temp (cons k' v a)
     | SConst, Temp m' ->
       Temp (cons k' v' (SAssoc m', d))
     | _ -> Unknown
    and n =
     Unknown
    in
    (match m.dstr (c, n) with
    | Temp m -> SAssoc m
    | _ -> SAssoc (cons k v a))
  | _ ->
    SAssoc (cons k v a)),
  _App (_Var "store", [snd k; snd v; snd a]))),
 _Var "store"
```

図 6　キーの重複の除去をともなう store
Fig. 6　store with removal of key-duplication.

る．以下が実行結果となる．

```
# print (conv (% let a = store (1, ‘x, null) in
                 let a = store (2, ‘y, a) in
                 let a = store (1, ‘z, a) in
                 foreach (fun (k, v, _) -> ‘echo v, (), a) %));;
```

```
let foreach =
 SFun (fun [c0; n0; a] ->
  match a with
  | SAssoc m, _ when m.ordered ->
    let c (k, v, _, Ans (x, r)) = mkans r (% c0 (k, v, x) %)
    and n = Ans (n0, empty)
    in
    let Ans (x, r) = m.dstr (c, n) in
    x, r
  | _ ->
    eval (% `foreach (c0, n0, a) %)),
 _Var "foreach"
```

図 7　foreach 操作
Fig. 7　foreach operation.

```
let _27 = store (1, x, null) in
let _41 = store (2, y, _27) in
let _48 = store (1, z, _41) in
let _53 = echo z in
let _54 = echo y in
_54
```

結果として foreach は正しく展開されている．走査の順序が各キーがはじめて追加された順になっていることに注意する．これは PHP で定める走査の順序と同じである．他の言語の連想配列でも同じような走査の順序が定められていることが多い．

ところで，上の結果では foreach が除去できたことでもうすでに store も使わなくなっている．次節ではこのような使わない let-定義を取りのぞくための手法を述べる．

3.3　遅延評価の利用と不要定義の除去

図 1 の評価器の性能は実はあまり良くない．理由は明白で HAbs のルールのところで eval が 2 回ずつ呼ばれる可能性があり組合せ爆発の危険があるからである．しかしこれを直すのは実は容易である．必要のない eval を遅延評価によって呼ばないようにすればよい[⋆1]．いま s : static 側はもとからクロージャを生成して，eval はクロージャが使われるまで呼ばれないのだから，遅延は d : dynamic 側に入れればよい．

具体的には OCaml の Lazy モジュールを利用して，dynamic 型の定義を下のように変

⋆1 このような手法の存在は Sumii ら[7] が言及しているが，彼らは結局このアプローチはとらなかった．

```
open Lazy
...
let dyn sv = force (snd sv)
let _Var x = lazy (Var x)
let _App (d, ds) = lazy (App (force d, map force ds))
let _BoolConst b = lazy (BoolConst b)
...
let rec eval e =
 match e with ...
 | HAbs (n, h) -> ...
   let d = lazy (...) in ...
```

図 8　遅延評価の利用のための改変
Fig. 8　Using lazy-evaluation.

える．

```
... and dynamic = exp lazy_t
```

さらに関連する部分のコードを**図 8** のように変更する．

さて，dynamic 値の遅延を導入したのはもう 1 つわけがある．それは不要定義の削除である．このためには，各 let–文による変数定義のところで，その変数が使われるかどうかを調べればよい．変数の使用を調べるために，いま導入した遅延を使うことができる．

図 1 の評価器で slet 関数は，コードのコピーや，副作用の順が変わることを避けるために let-文の挿入を行っていた．これを改良して新しい slet2 を以下のように定義する．

```
let slet2 r sv =
  let x = fresh () in
  let uc = ref false in
  if fst sv = SConst then sv, r else
  (fst sv, lazy (uc := true; Var x)),
  fun cont ->
    if not !uc then r cont
    else r (App (Abs ([x], cont), [dyn sv]))
```

slet2 の定義中，もし cont の中に変数 Var x がまったく現れないならば，使用フラグ !uc が false になるので，この手法でうまくいく．またこのケースで遅延されている dyn sv の評価も省略できるので，評価器自体のコストの削減にもつながっている．

定義した変数が使われないケースで削除できるのは，副作用のない let-文だけなので，

slet と slet2 は使いわける必要がある．副作用のないたとえば 図 6 の store 操作について，その返り値を slet の代わりに slet2 で作るように変更する．

以下が実行例である．

```
# print (conv (% let a = store (1, ‘x, null) in
                 let a = store (2, ‘y, a) in
                 let a = store (1, ‘z, a) in
                 foreach
                  ((fun (k, v, _) -> ‘echo v), (), a) %));;
let _3 = echo z in
let _4 = echo y in
_4
```

前節と同じ例であるが，まず遅延の導入自体によって eval 自体の処理のコストが削減されたことが，fresh で生成された変数名のカウントが小さくなっていることから推察できる．また，使用フラグの導入により，不要なストアが除去された．

3.4　コンパイル時配列の実体化

3.2 節ではコンパイル時配列の中で同じキーに対する重複した操作を取りのぞくような仕組みを導入した．しかしこれによって，残余コードの store の重複が取れるわけではない，つまり現時点で 2 節の目標 4 がまだ達成されていない．いまは配列の dynamic 値が必要になるケースで slet によって残余コードに残されたストア列を使う実装になっているからである．

NBE 評価器や型駆動の部分評価器において実体化（reification）[1),2),6)] とは，static の値から dynamic のコードを作り直す操作のことである．本稿の文脈でも SAssoc 値からコードへの変換をする reify 処理を考案することができる．**図 9** に reify 処理を定義する．コードは余計な let の挿入を省くために，若干煩雑になっているが，実質は SAssoc 値のリストをたどりながら store 操作をコードに出力する処理である．今回は m.dstr は (dynamic * residual) ans 型の蓄積パラメータを計算し，その中の residual と dynamic とがそれぞれ作られた store 操作の列とその結果値を表す．定義中の m.rc についてはこれから説明する．

このような実体化の副作用として，重複したストアが残余コードから取りのぞける．なぜならコンパイル時配列中ではすでにキーの重複が解消されているからである．しかし，実体化には利点だけがあるわけではない．

```
let reify sv =
 match fst sv with
 | SAssoc m when !(m.rc) < 2 ->
   let c (k, v, a, ans) =
    let d, r =
     match fst a, ans with
     | SAssoc m, Temp (d, r) when !(m.rc) < 2 ->
       d, r
     | _ ->
       dyn a, empty
    in
    match d with
    | Var _ ->
      Temp (App (Var "store", [dyn k; dyn v; d]), r)
    | _ ->
      let x = fresh () in
      Temp (App (Var "store", [dyn k; dyn v; Var x]),
            fun cont -> r (App (Abs ([x], cont), [d])))
   and n =
    Temp (Var "null", empty)
   in
   let Temp (d, r) = m.dstr (c, n) in
   d, r
 | _ ->
   dyn sv, empty

let slet2 r sv = ...
    else
      let d, r' = reify sv in
      r (r' (App (Abs ([x], cont), [d])))
```

図 9　コンパイル時配列の実体化
Fig. 9　Reification of compile-time arrays.

```
  let a = store (1, ‘x, ‘a) in
  let b = store (2, ‘y, a) in
  let _ = print_r a in
  let _ = print_r b in ()
```

たとえば上のようなコードでは配列 a，b がそれぞれ別に実体化されることで 1 行目の store(1, ‘x, ‘a) が複製されてしまい，逆にストア処理のコストが増加する可能性がある．

この問題には，コンパイル時配列のリストの各セルに参照カウントを追加し，複数箇所

```
type assoc =
    { rc : int ref; ... }
...
let inc rc = rc := !rc + 1

let rec eval e =
 match e with ...
 | HApp (e, es) -> ...
  match s with
  | SNone ->
    let args, r' =
     fold_right
      (fun sv (args, r') ->
       match sv with SAssoc m, _ ->
        (inc m.rc;
         let sv, r = slet2 empty sv in
         sv :: args, comp r r')
       | _ -> (sv :: args, r'))
      args ([], empty)
    in
    let r = comp r r' in
    ...

let nil = { rc = ref 2; ordered = true; dstr = (fun (c, n) -> n) }

let cons k v a =
 match fst a with
 | SAssoc m ->
   (inc m.rc;
   { rc = ref 0;
     ordered = m.ordered && fst k = SConst;
     dstr = (fun (c, n) -> c (k, v, a, m.dstr (c, n))) })
 | _ ->
   { rc = ref 0; ordered = false;
     dstr = (fun (c, n) -> c (k, v, a, Unknown)) }
```

図 10　コンパイル時配列の参照カウント操作
Fig. 10　Reference counting for compile-time arrays.

から参照されている配列は実体化しないという回避策がある．**図 10** のように assoc 型のレコードメンバに参照カウントのフィールド rc : int ref を追加し，さらに cons 操作でこのカウントを増やすようにする．またリストの先頭のセルの参照は eval の定義の HApp の引数のところで数えるようにしている．最後に図 9 の reify 関数ではリストの各セルを実体化するかどうかの判断のときに，この参照カウント m.rc を調べこれが 2 以上ならば store の作り直しをやめる．

動作は以下のようになる．

```
# print (conv (% let a = store (1, ‘x, null) in
                 let a = store (2, ‘y, a) in
                 let a = store (3, ‘z, a) in
                 let b = a in
                 let a = store (2, ‘p, a) in
                 let b = store (2, ‘q, b) in
                 let _ = ‘print_r a in
                 ‘print_r b %));;
let _0 = store (1, x, null) in
let _10 = store (2, p, _0) in
let _5 = store (3, z, _10) in
let _6 = print_r _5 in
let _9 = store (2, q, _0) in
let _7 = store (3, z, _9) in
let _8 = print_r _7 in
_8
```

ここでは最初の store(1, ‘x, null) のところのリストのコンスセルの参照カウントが 2 になっているので，結果コードでこの操作は slet によってソースコードと同じ場所に出ている．他の store は reify 関数が出力しているため，配列が使われる場所まで遅延されている．

このような手法で実際にどの程度コストの増加・減少があるかについては議論の余地がある．たとえば上の例では，最適化前後でキーの重複はなくなっているが実は全体のストアの数は変わっていない．コストの減少を確実に保証するためには，より複雑な解析やヒューリステクスなどが必要だと考えられるが，その場合でも let-insertion と実体化を組み合わせて store の出力を制御する手法は基本的なアイディアとなりうる．

3.5　短絡融合による最適化

配列を作る操作はストア操作だけではない．たとえば PHP の explode($ch, $s) 関数は

```
let err = lazy (failwith "[error]")
let do_c c k v a =
  let Ans ((s, d), r) =
   c (k, v, (SNone, err), Ans (a, empty))
  in
  let x = SFun (fun [] -> (s, d), r), err in
  (% x () %)
let do_n n =
  let Ans ((s, d), r) = n in
  let x = SFun (fun [] -> (s, d), r), err in
  (% x () %)

let explode =
  SFun (fun [ch; s] ->
    let dstr (c, n) =
      mkans empty
        (% `fix
            (fun f -> fun (a, pos, cnt) ->
             let npos = `strpos (s, ch, pos) in
             if `not `is_int npos then a
             else
               let v = `substr (s, pos, npos) in
               let a = { do_c c cnt v a } in
               f (a, npos, `add (cnt, 1)))
            ({ do_n n }, 0, 0) %)
    in
    slet2 empty
     (SAssoc { rc = ref 2; ordered = true; dstr = dstr },
      _App (_Var "explode", [snd ch; snd s]))),
  _Var "explode"
```

図 11　explode 操作
Fig. 11　explode operation.

文字列 $s を区切り文字 $ch で区切って部分文字列の配列にする関数である．今回の枠組みはこのような他の種類の配列の生成操作にもすぐに対応できる．

いま explode を文字列をパースしながら，配列を作る操作として**図 11** のように定義する．この定義でたとえば以下のような結果が得られる．

```
# print (conv (% let a = explode (",", `input) in
                 foreach ((fun (k, v, _) -> `echo v), (), a) %));;
let _1 =
 fix
  (fun _3 (_6, _5, _4) ->
   let _7 = strpos (input, ",", _5) in
   let _8 = is_int _7 in
   let _9 = not _8 in
   let _10 = if _9 then _6
    else
     (let _11 = substr (input, _5, _7) in
     let _12 = echo _11 in
     let _13 = add (_4, 1) in
     let _14 = _3 (_12, _7, _13) in
     _14) in
   _10) in
let _2 = _1 ((), 0, 0) in
_2
```

すなわち，配列を作ることなく文字列のパースから直接結果を出力するようなプログラムが得られた．ただし，strpos(s, c, f) は文字列 s 中 f 文字目以降で最初に文字 c が出現する場所を表すが，c が出現しない場合は () を返すとする．substr(s, f, t) は文字列 s の f 文字目から t 文字目までの部分文字列を計算するとし，また fix は不動点オペレータである．

ここで実現されているのは短絡融合（shortcut-fusion [5]）変換と呼ばれる関数融合の手法の一番基本的なもので，Haskell のコンパイラでは非常によく使われている．短絡融合変換のアイディアは，プログラム中のリストの構築子の組 (c, n) を，リストの消費者が使う破壊子，たとえば fun(k, v, _) -> `echo v と () で置きかえてしまうことでリスト生成をなくしたコードを出力することである．

コンパイル時の連想配列がすでにチャーチ符号表現になっているために，上のように explode を定義するだけで，この最適化ができた[⋆1]．

⋆1 ただし，現状のコードは explode で作った配列に対しての store 操作があると動作しない．これについては assoc 値に explode で定義されたことのフラグをつけて，これに対するストアは，つねに ordered ではない配列を作るという処理を入れるなどする必要がある．

4. 議　　論

4.1 命令型言語のコンパイラとの比較

本稿で提案した最適化のいくつかはデータフロー解析や書き換えに基づく命令型言語のコンパイラですでに解かれているような問題に見えるかもしれない．確かに load の除去に関しては，コピー伝播などを使えば[⋆1]，同じようなことが可能だと考えられる．

重複した store のケースはどうだろうか．命令型のコンパイラにある不要定義除去などが使えそうだが，そう簡単ではない．

```
let a = store (1, ‘x, null) in
let a = store (2, ‘f (), a) in
let a = store (1, ‘g (), a) in
a
```

PHP の連想配列で上のようなコードの 1 行目の store(1, ‘x, null) を除去することは正しくない．理由は 3.2 節で述べた要素の追加順序であり，このケースで foreach 節で a の要素を見ていく場合には，(1, g() の結果)，(2, f() の結果) という順でなくてはならない．もちろん，このケースで 3 行目を 1 行目に移動することも許されないので，問題は簡単ではない．

```
let _1 = f () in
let _3 = g () in
let _5 = store (1, _3, null) in
let _6 = store (2, _1, _5) in
_6
```

実体化に基づく本稿のやり方なら，期待どおり上のような結果が得られる．

また foreach の展開や，短絡融合のような最適化は普通の命令型コンパイラにはない．単一の手法かつごく小さなコードでこのような多様な最適化をまとめて行えるところが評価による最適化の魅力である．

4.2 関 連 研 究

NBE 評価や部分評価の手法の実用化にむけての取り組みをいくつかあげる．

Sumii ら[7] は本稿とほぼ同じ評価器から議論を始めているが，最終的に提案している評価器では不要な let–定義の除去やそのインライン展開のために，静的解析による使用カウントの見積りを使っている．しかし静的解析は使用カウントを過大に見積もる可能性がある．たとえば

```
let a = store (1, true, ‘z) in
if load (1, a) then () else a
```

など eq(1, 1) や if の条件が成立するという情報がなければ 1 行目の store 操作が除去できなくなるケースがありそうである．本稿のように部分評価中に必要に応じて使用カウントを見積もれば，上の例の store などは残余コードには残らない．一方，Sumii らの手法が良い結果コードを与えるケースもあるので，より詳細な比較が必要である．

Danvy ら[3] は局所変数の読み書きなどを含むような言語のインタプリタの部分評価によるコンパイル手法を議論している．このようなインタプリタの部分評価というのは実行時環境を壊して対象言語のローカル変数にすることなので，連想配列を壊すという本稿の問題に近い部分がある．ただし，インタプリタのプログラムはすべて部分評価器に与えられるはずなので，プログラムの一部しか得られないような本稿で扱っている問題では結果が違う部分があると思われる．

また，Filinski[4] や Lindley[6] は計算的（computational）λ 計算のような副作用を含めた言語に対する NBE 評価の手法を調べた．彼らは NBE 評価での let-insertion の方法をいくつか議論しているが，本稿の方法はそのなかで蓄積モナドに基づく手法だと考えられる．Lindley はさらに SML.NET コンパイラの上で NBE 評価による最適化を実験・評価した．

竹辺ら[10] は，部分評価を用いて PHP プログラムを高速化するシステム PHP-Mix を開発した．PHP-Mix はどちらかというと SQL を使った DB アクセスの部分などを静的に事前評価することなどに注力しているようである．公開されている実装では定数畳み込み以外の連想配列の最適化はあまり行われていないようであった．

5. ま と め

本稿では PHP などの連想配列処理を評価によって最適化するためのいくつかの手法を与えた．配列の走査などがあって，いままでの命令型言語コンパイラの書き換えに基づく手法ではすぐにはうまくいかないような最適化の問題をシンプルな手法で解決することができた．

今回の手法を PHP 言語全体などより広い範囲へ拡張していくとき，直近の問題としては，

⋆1 たとえば $a[0] = $x などで，配列の要素 $a[0] も変数と見なしこの値が $x のコピーであるという情報を次の $a[0] の参照まで伝える．

参照とエイリアスの問題がある．エイリアスを含むような連想配列に対しては，たとえば

```
function f($a) {
 $a[0] = 0;
 $a[1] = 1;
 echo $a[0];
}
```

のような場合のロード操作さえ除去してはならない．よってナイーブには解析でエイリアスを含みうるような連想配列を見つけ，これらをすべて動的なものとして扱わなくてはならない．

より良い方法として，副作用の影響も評価できるような評価器を考えている．これは参照だけでなく，実行系の内部状態を考慮に入れたような最適化にも有用だと考えられる．

謝辞　本稿のドラフトについて末永幸平氏および南出靖彦氏から多数のコメントをいただきました．また査読者の方には丁寧な査読をしていただきました．ありがとうございました．

参 考 文 献

1) Berger, U. and Schwichtenberg, H.: An Inverse of the Evaluation Functional for Typed lambda-calculus, *LICS*, pp.203–211 (1991).
2) Danvy, O.: Type-directed partial evaluation, *The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1996*, St. Petersburg Beach, Florida, pp.242–257 (1996).
3) Danvy, O. and Vestergaard, R.: Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation, *8th International Symposium on Programming Language Implementation and Logic Programming*, pp.182–197 (1996).
4) Filinski, A.: Normalization by Evaluation for the Computational Lambda-Calculus, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, Krakow, Poland, pp.151–165 (2001).
5) Gill, A., Launchbury, J. and Peyton Jones, S.L.: A short cut to deforestation, *FPCA '93: Proc. Conference on Functional programming languages and computer architecture*, New York, NY, USA, pp.223–232, ACM (1993).
6) Lindley, S.: Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages, Ph.D. Thesis, University of Edinburgh, College of Science and Engineering, School of Informatics (2005).
7) Sumii, E. and Kobayashi, N.: A Hybrid Approach to Online and Offline Partial Evaluation, *Higher-Order and Symbolic Computation*, Vol.14, No.2-3, pp.101–142 (2001).
8) Tozawa, A., Tatsubori, M., Onodera, T. and Minamide, Y.: Copy-on-write in the PHP language, *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, Savannah, GA, pp.200–212 (2009).
9) Tozawa, A., Tatsubori, M., Trent, S., Suzumura, T. and Onodera, T.: P9: High Performance PHP Runtime, 日本ソフトウェア科学会第 25 回大会予稿集 (2008).
10) 竹辺靖昭，湯淺太一：部分評価器を応用した動的 Web ページのキャッシュ機構，情報処理学会論文誌：プログラミング，Vol.43, No.SIG8 (PRO15), pp.98–109 (2002).

(平成 21 年 5 月 8 日受付)
(平成 21 年 8 月 21 日採録)

戸澤　晶彦（正会員）

1973 年生．2000 年東京大学大学院理学系研究科情報科学科専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にて，プログラミング言語やその応用の研究に従事．現在，同研究所リサーチャー．ACM，日本ソフトウェア科学会各会員．

小野寺民也（正会員）

1959 年生．1988 年東京大学大学院理学系研究科情報科学専門課程博士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にて，オブジェクト指向言語の設計および実装の研究に従事．現在，同研究所シニア・テクニカル・スタッフ・メンバ．インフラストラクチャ・ソフトウェア担当．第 41 回（平成 2 年後期）全国大会学術奨励賞，平成 7 年度山下記念研究賞，平成 16 年度論文賞，平成 16 年度業績賞，各受賞．理学博士．ACM Senior Member．日本ソフトウェア科学会会員．