情報処理学会論文誌　プログラミング　Vol.13　No.3　1–12 (June 2020)

Lift中間言語における動的長配列の追加

新美 和生[1,a)]　増原 英彦[1,b)]

受付日 2019年11月17日, 採録日 2020年3月20日

概要：Lift IL は GPGPU プログラムのための配列指向の中間言語である．Lift コンパイラは高級言語プログラムを OpenCL プログラムに変換する際に，中間言語である Lift IL 上でハードウェア依存の最適化を行う．一方，Lift IL には動的に長さが決まる配列を表現できないという問題がある．なぜならば Lift IL は静的に型付けされる関数型言語で，配列の長さをシンボルとして推論する依存型システムを備えており，静的にすべての配列長のシンボルが決定しているからである．本論文では，Lift IL を存在型で拡張することで動的長配列を扱う方法を提案する．その際，型から配列長を隠蔽・展開する操作を適切な場所に挿入する必要があるが，それらの自動挿入方法を提案する．この提案に基づいて，Lift IL のコンパイラを作成し，そのうえでフィルター関数を実装した．さらに，Project Euler 問題集を用いて，拡張した Lift IL コンパイラで動的長配列を扱うプログラムが記述・コンパイル・実行できることを確かめた．

キーワード：GPU 計算，コード生成，型理論，存在型，Lift

Extending Lift Intermediate Language with Dynamic Length Arrays

Kazuki Niimi[1,a)]　Hidehiko Masuhara[1,b)]

Received: November 17, 2019, Accepted: March 20, 2020

Abstract: Lift IL is an array-oriented intermediate language (IL) for GPGPU programming. The Lift compiler translates a high-level language program into Lift IL, applies hardware-specific optimizations, and then generates an OpenCL program. The IL has a dependent type system that infers the length of arrays as a symbol. This forces every array to have a length that can be statically determined, preventing one from expressing arrays having a dynamic length. In this paper, we extend Lift IL to support arrays having a dynamic length by using existential types. We also propose an algorithm that automatically inserts pack and unpack operations, which are necessary for existential types. Based on the proposal, we implemented a Lift IL compiler and added a filter function. In order to assess the usefulness to real world problems, we confirmed the problems that require the dynamic length array in a subset of Project Euler can successfully be written, compiled and executed in the extended Lift IL compiler.

Keywords: GPU computing, code generation, type theory, existential type, Lift

1. はじめに

GPU（Graphic Processing Unit）は主に画像処理に用いられる計算ユニットである．これを汎用（GPGPU：General-Purpose computing on GPU），たとえば機械学習などの並列計算に用いることが多くなっている．

GPU プログラミングを行う際に用いる OpenCL にはパフォーマンスの移植性の問題がある．その原因は，GPU はハードウェアによってスレッド数やメモリなどの特性が異なり，最適なスレッド数やループ構造が異なるからである．対象のハードウェアに最適化されたプログラムはハードウェアの特性に依存することになる．そのプログラムを他のハードウェアで動かそうとすると最適なスレッド数やループ構造が異なるため，対象のハードウェアで動かすより非効率となることがある．OpenCL のような低レベルプログラムだとハードウェアが変わると書き換えになってしまう．StreamIt [1] や LiquidMetal [2] は対象のハードウェアごとに専用のカーネルの実装を用意することでこの問題

[1] 東京工業大学情報理工学院
School of Computing, Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[a)] niimi.k.ab@prg.is.titech.ac.jp
[b)] masuhara@acm.org

```
1  (lambda (xs: Array[Int, N]) (map (lambda (
       x) (* x 3)) xs))
```

図 1　Lift 高級言語のプログラム例
Fig. 1　An example program of the high-level Lift language.

```
1  ; (o f g h ...) は合成関数
2  (lambda (xs: Array[Int, N]) ((o
3    join
4    (mapWorkgroup
5      (o
6        joinVec
7        (mapLocal
8          (mapVec (lambda (x) (* x 3))))
9        (splitVec 4)))
10   (split 1024))
11   xs))
```

図 2　高級言語から変換された Lift IL プログラム
Fig. 2　A Lift IL program translated from the high-level language program.

の解決を試みたが，ハードウェアの変更が生じた際にカーネルを再び最適化ないし再開発する必要があるという問題がある．

Lift [*1] は別のアプローチで解決を試みたプロジェクトである．まず，ユーザが記述する DSL などの高級言語プログラムを map や reduce を持つ関数型中間言語である Lift 高級言語プログラムに変換する．次に，Lift 高級言語プログラムを Lift IL プログラムに変換しハードウェアに最適なプログラムを探索する．最後に Lift IL プログラムを OpenCL のプログラムに変換する．Lift IL とは中間言語から OpenCL プログラムに変換する際に用いられる GPGPU 向けのデータ並列な関数型 IL（Intermediate Language：中間言語）のことである．

ユーザが記述した，整数型の各要素を 3 倍する高級言語プログラムを Lift 高級言語プログラムに変換した例（**図 1**）とそこから変換され，対象のハードウェアに最適化された Lift IL プログラム（**図 2**，推論された型を一部示す），OpenCL プログラムの例（**図 3**）を示す [3]．図 2 の 9，10 行目の `(split 1024)` や `(splitVec 4)` はハードウェアごとに最適な数値が異なる．この例では図 3 の 9 行目のように 4 要素のベクトル演算を行うコード生成が行われている．

文献 [4] では高級言語プログラムでのデータ構造の表現力を広げるために Lift IL の拡張を行っている．このように Lift IL の拡張をすることは高級言語の表現力を広げることになり有意義である．

本論文では以下，2 章で研究の背景を述べ，3 章で現状の問題点を指摘する．これらをふまえ，4 章・5 章で動的長配列の設計と実装について述べる．6 章では評価について述べ，7 章で関連研究を紹介し，8 章でまとめる．

[*1] http://www.lift-project.org/

```
1  int4 mul3(int4 x) { return x * 3; }
2  kernel vectorScal( global int * in,out,
       int N){
3    for ( int i= get_group_id ; i < N/1024;
4        i+= get_num_groups ) {
5      global int * grp_in = in+(i*1024);
6      global int * grp_out = out+(i*1024);
7      for ( int j= get_local_id ; j <
           1024/4;
8            j+= get_local_size ) {
9        global int4 * in_vec4 =( int4 *)
             grp_in+(j*4);
10       global int4 * out_vec4=( int4 *)
             grp_out+(j*4);
11       *out_vec4 = mul3(*in_vec4);
12 } } }
```

図 3　Lift IL プログラムから変換された OpenCL プログラム
Fig. 3　An OpenCL program translated from the Lift IL program.

2. 背景

本章では研究対象となる Lift IL の型システムと配列の表現，本研究の提案に必要となる存在型の説明をする．また，実装する関数を説明するために OpenCL の GPU モデルも説明する．

2.1 Lift IL

Lift IL とは GPGPU 向けのスケルトン並列な純粋関数型 IL である．スケルトン並列とは map や reduce などの関数を組み合わせて並列化されたプログラムを作成する手法である．OpenCL プログラムに変換される際には map 関数の呼び出しが並列化されたコードが出力される．

2.1.1 実行モデル

Lift IL コンパイラは高級言語プログラムから変換された Lift 高級言語プログラムを入力とする．Lift 高級言語プログラムを Lift IL プログラムに変換し，対象のデバイスで最も実行速度が速いプログラムを探索する [3]．最後に Lift IL プログラムを OpenCL プログラムに変換する（**図 4**）．OpenCL プログラムを実行するためのホストプログラムはユーザが記述する．

2.1.2 型システム

Lift IL は型推論を静的に行う強い型付けの言語で，Lift IL から OpenCL へのコンパイル時にすべての式の型を決定する．ソースコード中には入力配列の型を除いて型注釈は存在しない．

OpenCL にコンパイルする際に処理する配列の長さが必要となるので Lift IL は配列型に依存型を利用している．依存型とは値に依存する型のことで，Lift IL では配列の

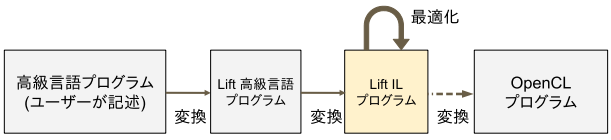


図 4　Lift IL の実行モデル（破線矢印は本研究の対象）
Fig. 4　The execution model of Lift IL (this research focuses on the translation shows as the dashed arrow).

```
1  (lift
2   (N) ; 入力されるサイズ型の変数
3   ((array-type float N)) ; 入力配列
4   (lambda (xs)
5    (o
6     join
7     ; 入力配列の型はArray[Array[Float, 2],
          N/2]
8     (mapSeq (lambda (ys: Array[Float, 2])
9      ; 入力配列の型はArray[Float, 2]
10      (mapSeq (lambda (y) (* y 2.0f)) ys)))
11    (split #2)
12    xs)))
```

図 5　配列を分割して 2 倍してから結合する Lift IL プログラム
Fig. 5　A Lift IL program that splits an array, doubles each element, and joins it.

長さのみが依存型となっている．たとえば要素が整数型（Int）で長さが変数 N に等しい配列型は Array[Int, N] と表される．配列の長さとして許されている式はサイズ型の変数と定数型どうしの演算である（例：42, N, N/2）．サイズ型は自然数の配列長を表す型である．

2 つの配列型が等しいかどうかは，要素の型が等しいことに加え長さの等しさを証明できるかどうかで判定される．たとえば Array[Int, N] と Array[Int, 2*N/2] は等しい．

2.1.3　配列の実行時の表現方法

Lift IL のカーネルが配列を引数として受け取るときはその配列長（**図 5** では引数の N）も配列長のシンボルと同じ名前の引数として受けとる．型推論により配列型の長さはシンボルと定数型で表すことができる．ゆえに，すべての配列型の長さは引数としてわたされた長さの変数と定数からなる式で表すことができる．

たとえば，図 5 は入力配列（要素の型は Float，長さは N）を 2 要素ずつに分割した後，それらを 2 倍する Lift プログラム（簡便のため一部省略）である．**図 6** はこれを OpenCL プログラムにコンパイルした結果である．コンパイル時に各配列の長さを表す式が分かっているので for 文の繰り返し数はその情報を利用している．図 5 の 8 行目の mapSeq の入力値の型は Array[Array[Float, 2], N/2] なので長さは N/2，10 行目の mapSeq の入力値の型は Array[Float, 2] なので長さは 2 となる．

```
1  kernel void KERNEL(float *xs, float *
       result, int N) {
2    ...
3    // 8行目のmapSeq
4    for (int i = 0; i < N / 2; i++) {
5      // 10行目のmapSeq
6      for(int j = 0; j < 2; j++) {
7        ...
8      }}}
```

図 6　図 5 のコンパイル結果
Fig. 6　Compiled code from Fig. 5.

構文

$term$ ::=...

pack $\{*T, t\}$ **as** T　　*pack*

unpack $\{X, x\}$ = t_1 **in** t_2　　*unpack*

図 7　pack と unpack の構文
Fig. 7　The syntax of pack and unpack.

2.2　存在型

存在型の一般的な定義と利用例を説明する．本節の定義と文法は文献 [5] の 24 章に従う．存在型とは型の一部の情報を隠蔽した型のことで，モジュールやオブジェクトの型として使われる．本研究では存在型を $\{\exists X, T\}$ と表す．ここで X は型変数，T は型であり内部に X の出現を許す．存在型の値を $\{*S, t\}$ と表す．S は型，t は T に出現する X を S に置き換えた型の項である．たとえば {*Int, λx:Int.succ(x)} は存在型 $\{\exists X, X \to X\}$ などを持つ．

pack は，存在型を導入するための構文である．as の左側の値の型を右側の型として扱うように型検査器に指示する（**図 7**）．なぜこの構文が必要かというと存在型の値が持つ存在型は一意に定まらないからである．たとえば値 {*Int, λx:Int.succ(x)} が持つ型は $\{\exists X, X \to X\}$ のほかにも $\{\exists X, Int \to X\}$ などが考えられる．存在型の値が与えられたとき，それがどの型なのか型検査器が推論するのは不可能なので，明示する必要がある．

unpack は，存在型の値を使うための構文である（図 7）．X と x を存在型の値 t_1 のそれぞれ対応する値に束縛して t_2 を評価する．**図 8** は pack と unpack を用いたサンプルプログラムである．t_2 である x.f x.a を評価する際には X は具体的な型 Int としては扱われず抽象的な型 a: X, f: X -> Int として扱われる．

2.3　OpenCL の GPU モデル

OpenCL は GPU などの**デバイス**という大きな単位がありその中に**ワークグループ**が複数あり，各ワークグループ

```
let t = pack {
      *Int, { a: 1, f: λx:Int. succ(x) }
      as {∃X, {a: X, f: X -> Int}}} in
  unpack {X, x} = t in
    ; x: { a: X, f: X -> Int }
    x.f x.a ; = 2:Int
```

図 8　一般的な存在型の使用例
Fig. 8　An example usage of existential types in general.

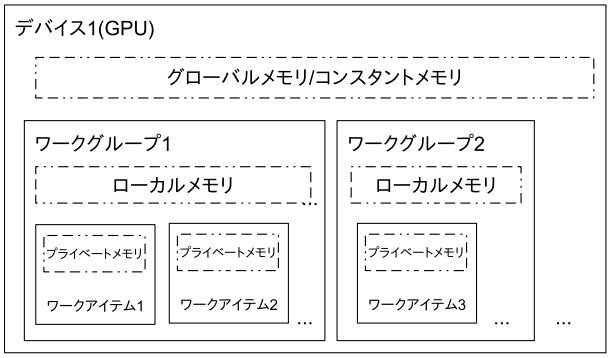


図 9　OpenCL のスレッドとメモリモデル [6]
Fig. 9　The threads and memory models of OpenCL.

内にスレッドに相当するワークアイテムが複数ある（図 9）．

各単位ごとに対応するメモリが存在し，デバイスにはグローバルメモリ，ワークグループにはローカルメモリ，ワークアイテムにはプライベートメモリがある[*2]．各単位はそれを包含する単位に対応するメモリにアクセスできる．たとえばワークアイテムはプライベートメモリ，ローカルメモリ，グローバルメモリにアクセスすることができる．

3. Lift IL の問題点

Lift IL では動的長配列を使用することができない．**動的長配列**とは Lift IL プログラム内に出現しない変数でのみ表現できる配列型であり，実行時に配列長が決定する可変長配列とは区別される．本章では動的長配列を用いたプログラム例を紹介し，それらが Lift IL で扱えない理由を述べる．

まず，動的長配列を扱うプログラム例を示す．プログラムは Γ `|- t` のように書いて，`t` に現れる自由変数の型環境 Γ 下で `t` を評価するということを表す．また，先行する例の Γ に現れた定義は省略する．

`filter` 関数は第 1 引数の関数（述語関数）が真を返す要素のみを集める関数である．

(1) 動的長配列を作る例 (filter)

```
xs:Array[Float,N],
filter:(A->Boolean)->Array[A,N]->
    Array[A,?]
f:Float->Boolean
|- (filter f xs)
```

生成される配列の長さは要素の値に依存するので，動的長配列が生成される．

(2) 複数の動的長配列を要素とする例 (map)

```
ys:Array[Array[Float,N],M],
map:(A->B)->Array[A,N]->Array[B,N]
|- (map (lambda (xs) (filter f xs))
    ys)
```

この式は配列の要素が (filter) の結果であるため，それぞれが異なる長さを持つ動的長配列を要素に持つ配列を作成する．

(3) 2 つの動的長配列を用いる例 (zip)

```
ys:Array[Array[Float,N],M]
zip:Array[A,N]->Array[B,N]->Array[(A
    ,B),N]
aref:Array[A,N]->Size->A
|- (let zs (map (lambda (xs) (filter
     f xs)) ys)
     (zip (aref zs 0) (aref zs 1)))
```

この式は (map) の結果の 1 番目と 2 番目の要素に対して zip をする．これらは長さが異なりうるので型付けできない，誤ったプログラムである[*3]．

(4) 動的長配列と等しい長さの配列を作る例 (derive)

```
xs:Array[Float,N],
g:Float->Float
|- (let xs' (filter f xs)
     (zip (map g xs') xs'))
```

この式は動的長配列に対して zip を行っているが，その相手は自身と同じ長さであることが分かるので，正しいプログラムである．

現在の Lift IL の型システムでは動的長配列を扱うことはできない．その理由は以下のとおりである．Lift IL では配列長にはプログラム中にすでに出現しているサイズ型の変数を用いた式に限定しており，要素の値に依存して長さが決まるような配列には型を与えられないからである．

型システムの単純な拡張はたとえば以下の方針が考えられるが，これらには問題がある．

(a) 動的長配列を新たに作る式があったら，他の配列長とは異なる長さになるように新たに長さ型の変数を導入する．

(b) 新たに動的長配列型を用意し，この型を持つ配列は長さが未知とする．

まず，(a) を採用した場合，(map) のような動的長配列を要素とする配列を表現することができない．もし，(map) の結果の型を `Array[Array[Float, N'], M]` としてしま

[*2] デバイスにはグローバルメモリの他にコンスタントメモリが存在するが本論文では割愛する．

[*3] 本論文では let 式を `(let x t1 t2)` と表す．`t1` を評価した結果を `x` に束縛し追加した環境で `t2` を評価する．

うと (zip) の誤りを検出できなくなってしまう．(b) を採用した場合，(derive) のように動的長配列と等しい長さの配列を等しく扱うことができない．

4. 提案：存在型を用いた動的長配列の型付け

本研究では Lift IL を存在型で拡張し，配列の長さを隠蔽することで動的長配列を表現する．それにより，長さに関する同一性を判定できるような式を制限する．一般的な存在型を扱う際に明示的に記述される pack と unpack を自動挿入する方法も提案する．

なお，以下では型規則と変換規則を提案するが，これらに関する健全性の証明は行っていない．

4.1 動的長配列のための構文

動的長配列の型を「長さを隠した配列型」として表す．つまり，動的長配列の型を $\{\exists X, Array[T, X]\}$ のように表す．フィルター関数のように動的長配列を作る関数の返り値の型はこの型を付ける．次に動的長配列を扱うための構文 pack と unpack を導入する．これらは最終的には自動挿入されるが，型検査の規則を説明するために導入する．

pack は，動的長配列の導入規則を適用する場所を指定する（図 10）[*4]．たとえば xs の型が $Array[Int, N]$ の場合，(**pack** xs) の型は $\{\exists X, Array[Int, X]\}$ となる．図 7 の as の右側に相当する型指定がないが，これは t の型は $\{\exists X, Array[T, X]\}$ という形をとることが明らかなので省略している．

unpack は，動的長配列型の除去規則の適用場所を指定する（図 11）．具体的には動的長配列 $\{\exists N, Array[I, N]\}$ である t_1 を長さを X として x（型は $Array[I, X]$）に束縛して，t_2 を評価する[*5]．

本項で導入した動的長配列を用いてフィルター関数の型は以下のように表される．

$$filter :: (A \to Boolean) \to Array[A, N] \to \{\exists X, Array[A, X]\}$$

このフィルター関数（filterSeq）を用いたプログラムのサンプルを型付け結果とともに掲載する（図 12）．

現在は最も外側の配列長のみを隠蔽することを許しているが今後は {∃X, Array[Array[Float, X]]} のような形の存在型も必要性を含めて検討する．

4.2 型規則

Lift IL の型付け規則は先行研究 [3] で与えられたものである．本研究ではこれらの規則に図 13 の型付け規則を加える．

[*4] 全体の構文は付録 A.1 で示す．
[*5] この形式では取り出した配列の長さを扱うことができないため，将来的には unpack で長さの変数を指定できるようにする．

構文 (項)

$term$ ::= ...

(**pack** t)　　*pack*

図 10　pack の項
Fig. 10　The syntax of pack.

構文 (項)

$term$ ::=...

(**unpack** x t_1 t_2)　　*unpack*

図 11　unpack の項
Fig. 11　The syntax of unpack.

```
1  (lift (N) ((array-type float N))
2   (lambda (xs)
3    (unpack ys: Array[Float, X1]
4      (filterSeq
5        (lambda (x) (< x 0.5f)) xs)
6          : {∃X, Array[Float, X]}
7     (pack
8       (mapSeq (lambda (x) (* x 2.0f)) ys)
9         : Array[Float, X1]
10      ): {∃X, Array[Float, X]})))
```

図 12　0.5 未満の数値を抽出しそれを 2 倍する Lift プログラム（一部省略）
Fig. 12　A Lift program that extracts numbers less than 0.5 and double them (extract).

型付け規則

$$\frac{\Delta; \Gamma \vdash t : [X \mapsto N]Array[I, X]}{\Delta; \Gamma \vdash (\textbf{pack}\ t) : \{\exists X, Array[I, X]\}}$$

$$\frac{\Delta; \Gamma \vdash t_1 : \{\exists N, Array[I, N]\} \quad \text{X is unique} \quad \Delta, X; \Gamma, x : Array[I, X] \vdash t_2 : T_2 \quad FV(T_2) \cap \{X\} = \phi}{\Delta; \Gamma \vdash (\textbf{unpack}\ x\ t_1\ t_2) : T_2}$$

ただし $FV(T)$ は T の自由変数である．

図 13　pack と unpack の型付け規則
Fig. 13　Typing rules for pack and unpack.

unpack の型付け規則は，unpack 時に決めた配列の長さが t_2 の型に自由出現しないことを求めているので，unpack した配列長が含まれる配列は pack する必要がある．もちろん，reduce などの関数で配列の長さをなくしてしまえば pack する必要はなくなる．

自由出現を許さない理由は以下の誤りを防ぐためである．図 14 は 2 次元配列の各要素についてフィルターするプログラムである．unpack は動的長配列から長さを取り出す操

```
1  xss: Array[Array[Float, N], M] |-
2    (map
3      (lambda (xs) ; この関数の型は不定
4        (unpack xs' (filterSeq f xs') xs'))
5      xss)
```

図 14　2 次元配列の各要素についてフィルターするプログラム
Fig. 14　A program that filters elements in an array of arrays.

$$\frac{\Gamma \vdash \texttt{(filterSeq (...) xs)} : \{\exists X, Array[Float, X]\} \qquad D}{\Gamma \vdash \texttt{program} : \{\exists X, Array[Float, X]\}}$$

$$D = \frac{\dfrac{\Gamma, X_1, ys : Array[Float, X_1] \vdash \texttt{(mapSeq (...) ys)} : Array[Float, X_1]}{\Gamma, X_1, ys : Array[Float, X_1] \vdash \texttt{(pack (mapSeq (...) ys))} : \{\exists X, Array[Float, X]\}}}{}$$

ただし $\Gamma = xs : Array[Float, N]$

図 15　図 12 の導出木
Fig. 15　The derivation tree of Fig. 12.

作に相当するが，同じ unpack 式を複数回実行して得られる長さは等しいとは限らない．ゆえに，もし自由出現を許したとしたら図 14 の 3 行目のラムダ式の型は不定となってしまう．本提案では unpack で取り出した長さの変数の自由出現を禁じることにより，unpack 式が複数回実行されても 1 回目と 2 回目の長さが同時に現れることはない．

具体例として，図 12 の導出木を**図 15** に示す（対象のプログラムを `program` とする）．

4.3　動的長配列の型検査

以上の規則を用いて，3 章であげたプログラムが正しく型付けされることを示す．

(filter) を `pack`，`unpack` を使うように書き換え，型付けしたものを**図 16** に示す．このプログラムは正しく型付けされている．

(map) も同様に書き換え，型付けしたものを**図 17** に示す．このプログラムも正しく型付けされている．

(zip) も同様に書き換え，型付けしたものを**図 18** に示す．このプログラムは `zs` の要素を `zs1` と `zs2` に unpack して束縛しているが同じ存在型でも unpack する際は必ず他の配列型とは長さが異なるものとして扱われるので `zs1` と `zs2` の長さは異なり，型誤りとなる．

(derive) も同様に書き換え，型付けしたものを**図 19** に示す．このプログラムは図 18 とは異なり，unpack した配列 `xs'` とそれを map 関数に適用した配列を zip しており，長さは同じなので正しく型付けされている．

4.4　pack，unpack の自動挿入

動的長配列を扱うのに必要となる pack と unpack の自動挿入を行う．pack と unpack をプログラマがそれを書くのは煩わしいので省略できるようにしたい．しかし，unpack は存在量化子を外すために必須で，pack は単に省略してしまうと無限に規則が適用できてしまい型検査が終わらなくなってしまう．直感的には unpack は動的長配列型が最初に出現した位置から関数の末尾まで，pack は unpack で決定された長さの変数を含むときのみ unpack の末尾に挿入される．以下に pack と unpack の自動挿入を行う方法を述べる．

(1) プログラムを K 正規化する．K 正規化とは構文木上で入れ子になっている式を変数として let 定義する変換のことである [7]．たとえば `(map g (filter f xs))` という式は `(let xs2 (filter f xs) (map g`

```
1  xs:Array[Float,N] |-
2    (filter f xs): {∃X, Array[Float, X]}
```

図 16　(filter) を書き換え，型付けしたプログラム
Fig. 16　A modified and typed version of (filter).

```
1  ys:Array[Array[Float,N],M] |-
2    (map (lambda (xs) (filter f xs)) ys)
3      : Array[{∃X, Array[Float, X]}, M]
```

図 17　(map) を書き換え，型付けしたプログラム
Fig. 17　A modified and typed version of (map).

```
1  ys:Array[Array[Float,N],M] |-
2    (let zs: Array[{∃X, Array[Float, X]}, M
         ]
3      (map (lambda (xs) (filter f xs)) ys)
4      (unpack zs1: Array[Float, N1] (aref zs
           0)
5        (pack (unpack zs2: Array[Float, N2]
6          (aref zs 1)
7          (pack
8            ; N1とN2は等しくないので型誤り
9            (zip zs1 zs2)
10         )))))
```

図 18　(zip) を書き換え，型付けしたプログラム
Fig. 18　A modified and typed version of (zip).

```
1  xs:Array[Float,N] |-
2    (unpack xs': Array[Float, N1]
3      (filter f xs)
4      (pack
5        (zip (map g xs'): Array[Float, N1]
6             xs'): Array[(Float, Float), N1]
7      ))
8    : Array[{∃X, Array[(Float, Float), X]},
         M]
```

図 19　(derive) を書き換え，型付けしたプログラム
Fig. 19　A modified and typed version of (derive).

$$\frac{\Gamma \vdash t_1 \rightsquigarrow t_1' : T_1 \quad \Gamma, x : T_1 \vdash t_2 \rightsquigarrow t_2' : T_2}{\Gamma \vdash (\texttt{let}\ x\ t_1\ t_2) \rightsquigarrow (\texttt{let}\ x\ t_1'\ t_2') : T_2}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow t_1' : \{\exists X, T_1\} \quad \Gamma, X, x : T_1 \vdash t_2 \rightsquigarrow t_2' : T_2 \quad FV(T_2) \cap \{X\} = \phi}{\Gamma \vdash (\texttt{let}\ x\ t_1\ t_2) \rightsquigarrow (\texttt{unpack}\ x\ t_1'\ t_2') : T_2}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow t_1' : \{\exists X, T_1\} \quad \Gamma, X, x : T_1 \vdash t_2 \rightsquigarrow t_2' : T_2 \quad FV(T_2) \cap \{X\} \neq \phi}{\Gamma \vdash (\texttt{let}\ x\ t_1\ t_2) \rightsquigarrow (\texttt{unpack}\ x\ t_1'\ (\texttt{pack}\ t_2')) : \{\exists X, T_2\}}$$

図 20　let を pack, unpack を含む式に置き換える規則
Fig. 20　The rewriting rules that replace let by pack and unpack.

xs2))という式に変換される．

(2) 図 **20** の書き換え規則により let を pack，unpack を含む式に置き換える．

5. 実現

拡張された Lift IL 処理系を以下のように作成した．まず，Lift IL コンパイラのサブセットの実装を論文 [8] の内容に従って Scala 言語で行った．さらに，その Lift IL コンパイラに言語要素として動的長配列，pack，unpack を追加した．さらに，動的長配列を返す関数の一例としてフィルター関数を実装した．

ソースコード，および評価に用いたプログラムは https://github.com/prg-titech/min-lift で公開している．

5.1 Lift IL コンパイラのサブセットの実装

実装した Lift IL コンパイラのサブセットはオリジナルの実装 [8] のうち，以下の機能を持つ．

- mapGlb, mapWrg, mapLcl, mapSeq, reduceSeq, split, join，toGlobal，toLocal，toPrivate
- 型解析
- メモリ割当て
- 多次元の配列アクセス
- OpenCL のコード生成

min-lift が生成するコードはオリジナルの実装と比べて配列アクセスの単純化などの最適化が施されてない．また，min-lift は slide や gather などの関数を実装していないので書くことができるプログラムの範囲はオリジナルの実装より狭い．

Lift IL コンパイラのサブセットは以下の順に処理を行う．

(1) 構文解析：字句解析した後，再帰下降構文解析で構文木を生成する．文法は構文解析の簡易化を図るため S 式を用いた．

(2) 型推論：型制約生成と単一化による教科書的な方法で行う（文献 [5] の 22 章）．

(3) 配列 View の作成：View Construction というある式が配列にどのような関数でアクセスしているか，というリスト形式の情報を View という．View を構文木をボトムアップに走査することにより生成し，View Consumption で実際の添字を View を走査することにより生成する（実際に生成するのはコード生成時）．Lift IL では配列のアクセスは暗黙のうちに行われるが，どの要素にアクセスするかをこの段階で決定する．split や join などで構造が変わる場合でも新たに領域を確保しないようにすることができる．

(4) メモリ領域の決定：Lift では関数で処理した結果を GPU 上のどのメモリ領域に格納するかを指定することができる．その情報を元に，どの式の結果がどのメモリ領域に格納されるかを再帰的に構文木を走査することで求める．

(5) コード生成：(4) までに求めた情報（型，View，メモリ領域）を元に，OpenCL のコードを生成する．

5.2 Lift IL の拡張

次に，本研究の目的である動的長配列を実現するために行った実装について述べる．

5.2.1 動的長配列，unpack，pack の追加

- **動的長配列**：型の定義，推論アルゴリズムを図 10，図 11，図 13 に従って拡張した．pack の自動挿入により $FV(T_2) \cap \{X\} = \phi$ の制約は自動的に満たされる．コード生成器は次のようにして動的長配列に対応させる．Lift IL プログラムから OpenCL へのコンパイル後でのプログラム中での動的長配列を，そのデータと長さを含んだ構造体で表現する．動的長配列にアクセスする際には動的長配列のデータを，その長さを使う際には動的長配列の長さを使うようにコード生成する．また，動的長配列を返す関数のコードの末尾に関数の結果の配列長を動的長配列の長さに設定するコードを生成する．動的長配列のデータは入力配列長分の大きさをあらかじめホストプログラム側で確保しておく．
- **unpack**：初めに，unpack は let から自動で書き換えるので，構文としては用意しない．次に，型検査器の制約を生成する処理に unpack の場合を追加する．`(unpack id value body)` を処理する場合を考える．
  (1) `value` の制約を生成する．
  (2) unpack で束縛される配列の一意の長さの変数を生成し，そこから配列型を生成する．
  (3) `id` と生成した配列型の組を新たな環境に追加し，

body の制約を生成する．

最後に，コード生成器の処理に unpack の場合を追加する．具体的には value を評価しその結果を id に設定し，動的長配列の構造体の長さを unpack で生成した配列長の変数に設定する．さらに body を評価するコードを生成している．

- **pack**：pack は自動挿入するので構文としては用意しない．型検査時は，pack する配列を存在型として扱うようにする．コード生成時は単に pack する対象の項をコード生成する．

5.2.2　フィルター関数の追加

Lift IL では，配列をどのようにワークグループやスレッドを使用しながら処理するかを関数の接尾語によって指定することができる．今回は各ワークアイテムが逐次にフィルター処理をする filterSeq とすべてのワークアイテムを使用する filterGlb を実装した．まず，**図 21** の定義を型検査器の初期環境に追加した．

次に，コード生成器の修正を行い filterSeq と filterGlb のコードを出力できるようにした．

フィルター処理は，各要素に述語関数を適用し残す要素を選択する，選択された要素の添字の計算，選択された要素の結果配列へのコピーからなる．本研究では 2 番目の選択された要素の添字の計算に prefix sum を並列に処理するアルゴリズム [9] を用いる．prefix sum とは各要素 x_i について i 番目までの和である．たとえば [1, 2, 3, 4, 5] に対する prefix sum は [1, 3, 6, 10, 15] となる．

5.3　未実現項目

以下は時間の都合上，実装できなかった項目をあげる．

- 動的長配列の OpenCL コードでの表現に構造体を利用するアイデア．現時点では長さを表す変数を別に用意している．
- pack と unpack の完全な自動挿入．unpack の自動挿入時に t_1 の型を決定しておく必要があるが，今回は簡略のため unpack の型検査時にそれまでの制約式と t_1 の制約式を用いて t_1 の型を決定している．その方法はすべての制約式を用いているわけではないので偽陽性の型誤りをしてしまう可能性がある．

5.4　コンパイル例

図 22 をコンパイルした OpenCL コードの内，filterSeq と動的長配列の長さの取り出しに対応している箇所を抜粋したものを**図 23**，ホストプログラムを**図 24** に示す．KERNEL は filterGlb までのコードとフィルター処理の「要素に述語関数を適用し残す要素を選択する」操作に相当する．KERNEL2 はフィルター処理の「選択された要素の結果配列へのコピー」と，動的長配列の長さの設定（22 行目）に相当する．unpack の動的長配列の長さを取り出す操作はホスト側で行われている．図 24 の 5 行目で長さの変数を取り出し，9 行目で KERNEL3 での長さの変数である N に設定している．

filterSeq:

$$\forall A.\forall B.(A \to Boolean) \to Array[A, B] \to \{\exists X, Array[A, X]\}$$

filterGlb:

$$\forall A.\forall B.(A \to Boolean) \to Array[A, B] \to \{\exists X, Array[A, X]\}$$

図 21　filterSeq，filterGlb の型定義
Fig. 21　The types of filterSeq and filterGlb.

```
1  (lift
2   (N)
3   ((array-type float N))
4   (lambda (xs)
5    ((toGlobal (mapGlb (lambda (x) (* x 2.0f
         ))))
6      (filterGlb (lambda (x) (> x 0.5f)) xs
          ))))
```

図 22　filterGlb を用いた Lift IL プログラム
Fig. 22　A Lift IL program that uses filterGlb.

```
1  kernel void KERNEL(
2      const global float* restrict xs,
3      global float* result,
4      global int* result_size,
5      global int* bitmap,
6      int N) {
7    int i1 = get_global_id(0);
8    bitmap[i1] = xs[i1] > 0.5f;
9  }
10
11 kernel void KERNEL2(
12     const global float* restrict xs,
13     global float* result,
14     global int* result_size,
15     global int* bitmap,
16     global int* indices,
17     int N) {
18   int i1 = get_global_id(0);
19   if (bitmap[i1]) {
20     result[indices[i1] - 1] = xs[i1];
21   }
22   *result_size = indices[N - 1];
23 }
24 kernel void KERNEL3(
25     const global float* restrict xs,
26     global float* result,
27     global int* result_size,
28     int N) {  ... }
```

図 23　図 22 のコンパイル結果（抜粋）
Fig. 23　Compiled code from Fig. 22 (excerpted).

```
1  // KERNEL2の実行後
2  int raw_result_size = 0;
3  // KERNEL2の引数result_sizeの値を
4  // raw_result_sizeにコピーする
5  queue.enqueueReadBuffer(result_size,
       CL_TRUE, 0, sizeof(int),
       reinterpret_cast<void*>(&
       raw_result_size));
6
7  // KERNEL3の引数Nに設定
8  cl_int cl_N = raw_result_size;
9  kernel3->setArg(arg_i, sizeof(cl_N), &cl_N
       );
10
11 // KERNEL3を実行する
```

図 24　図 23 を実行するホストプログラム（抜粋）
Fig. 24　The host program that invokes Fig. 23 (excerpted).

6. 評価

6.1 環境

評価に用いた環境は以下のとおりである．

- Ubuntu 18.04.3 LTS
- GeForce GTX 1080
- Intel(R) Core(TM) i5-6600
- OpenCL 1.2
  - コンパイルオプションは無指定（最適化は有効，strict aliasing は無効）
- GPU ドライバーバージョン：390.116

6.2 評価 1

本研究で提案した動的長配列と pack/unpack の自動挿入の有用性を示すために，拡張した Lift IL によって Project Euler [*6] の解法を見つけられる問題数が増えたことを示す．ここで解法とは，自然に並列化できるプログラムとする．Project Euler はプログラミングの問題集である．Project Euler には数列処理の問題が多く，Lift IL を用いて解くことができる問題がいくつかある．既存の Lift IL では解く方法を見つけられないが，本研究の提案手法により解法を見つけることができる問題数を確認する．本評価では動的長配列を返す関数の一例であるフィルター関数を追加する．また，そこで使われるはずだった pack と unpack の個数を調べることによりどれくらいの pack，unpack が省略されたかを確認する．

今回は Project Euler のアーカイブから最初の 30 問を抜き出した．それらの問題の中でフィルター関数を用いて解くことができる問題を探し，回答を Lift IL で記述，コンパイルできることを確認する．

[*6] https://projecteuler.net/

表 1　フィルター関数を用いて解法を見つけることができた問題
Table 1　The number of problem that can be found the solution by using a filter function.

問番号	省略された pack と unpack の組の個数	ファイル名
1	1	euler01.lisp
10	0	prime.lisp, prime2.lisp prime.cpp（100 以下）
30	1	euler30.lisp

その結果，既存の Lift IL で 4 問の解法を見つけ，さらにフィルター関数を追加することで 3 問の解法を見つけることができた（**表 1**）．表 1 のファイル名のプログラムは　https://github.com/prg-titech/min-lift/tree/master/advanced-examples にアップロードしている．検証した問題についてのリストは https://github.com/prg-titech/min-lift/blob/master/pro2019-4/project-euler-list.csv にアップロードしている．

また，フィルター関数を追加することで解くことができた 3 問の中で 2 組の pack と unpack が省略されたことが分かった．

評価を行う課程でフィルター関数の追加のみでは解法を見つけられない問題が 9 問あり，以下の機能を使うことができればこの 9 問の解法を見つけられることが分かった．

- **配列へのインデックスによるアクセス**：Lift IL では配列の n 番目の要素のみを取り出すという操作ができない．この操作ができるとたとえばソート処理を実装することができるようになる．
- **if 式**：たとえば，if 式と reduce を組み合せることにより配列の最大値を求めることができるようになる．
- **整数列の生成**：たとえば，n までの繰り返しや組合せ列挙ができるようになる．

pack と unpack の自動挿入については，解いた問題がフィルター関数を 1 つしか使わない問題のみであったので自動挿入の恩恵を受けにくいと考えた．さらなる評価としてフィルター関数を複数回使うような複雑な問題についても検証したいと考えた．

6.3 評価 2

本研究で提案したフィルター関数の実行時間が妥当であることを示すために，フィルター処理を含むプログラムを Lift IL と OpenCL でそれぞれ記述しカーネルの実行時間を比較した．さらに，プログラムの書きやすさを評価するために行数も比較した．

長さ 10,000 の各要素の範囲が $[0, 1)$ の浮動小数点数（float）配列について，0.5 より大きい値を抽出しそれに対して 2 倍するプログラムを Lift IL（`filter-gt-`

表 2　評価プログラムの実行時間（試行回数 100）
Table 2　The execution times of the benchmark program (number of trials: 100).

種類	スレッド数	平均（μs）	標準偏差（μs）
Lift IL	10,000	569.603	46.6280
OpenCL の並列	10,000	524.233	29.9401
OpenCL の逐次	1	7347.69	57.2716

表 3　評価プログラムの行数
Table 3　The sizes of the benchmark program.

種類	行数
Lift IL	6
OpenCL の並列	53
OpenCL の逐次	18

`0.5-twice-lift.lisp`[*7]），手で書いた OpenCL の並列（`filter-gt-0.5-twice-hand.cl`），手で書いた OpenCL の逐次（`filter-gt-0.5-twice-seq.cl`）で実装しそれぞれを 10 回実行したカーネルの実行時間の平均と標準偏差をとった（**表 2**）．また，それぞれのプログラムの行数を計測した（**表 3**）．カーネルの実行時間は `clGetEventProfilingInfo` 関数を用いて `CL_PROFILING_COMMAND_START` と `CL_PROFILING_COMMAND_END` の差分をとることで計測している．

この結果から分かることは逐次版は他の 2 つより約 13.4 倍の時間がかかっており，Lift IL 版は並列版の約 1.08 倍ということである．つまり，Lift IL 版は手で書いた並列な OpenCL プログラムと実行時間にほぼ差がないことが分かる．

プログラムの行数については，Lift IL 版は OpenCL の並列版に比べて約 1.1 割，OpenCL の逐次版に比べて約 3.3 割の行数で記述することができた．この結果からプログラムが書きやすくなっていることが分かる．

7. 関連研究

7.1 Existential Dependent Types

Xi らは実用的なプログラミングで依存型を使えるようにするために ML 言語を拡張した [10], [11]．彼らは既存の依存型付き言語の問題点として，フィルター関数の型を依存型で表現できないことをあげ，解決方法として *ML* 言語の Universal Dependent Type 拡張言語を Existential Dependent Types で拡張した $ML_0^{\Pi\Sigma}(C)$ を提案した．

min-lift と $ML_0^{\Pi\Sigma}(C)$ の構文と型規則を比較すると，後者は前者の一般化となっている．異なる点は，$ML_0^{\Pi\Sigma}(C)$ は任意の型を存在型にできることと，pack 構文の型規則において隠蔽される表現型の依存型の条件があることである．一方，min-lift においてその条件が不要である理由は，隠蔽される表現型がサイズ型のみが許されているからである[*8]．

```
1  data Vect : (len : Nat) -> (elem : Type)
       -> Type
```

図 25　Idris の Vect の定義
Fig. 25　The type of Vect in Idris.

```
1  filter : (elem -> Bool) -> Vect len elem
       -> (p : Nat ** Vect p elem)
```

図 26　Idris の filter の定義
Fig. 26　The type of filter in Idris.

7.2 Idris

プログラミング言語 Idris [12] は依存型を導入している純粋関数型プログラミング言語である．Idris には Vect という固定長リストが存在し型に長さが含まれている（**図 25**）．そして Vect に対する filter が存在し，その返り値の型に存在型を導入している（**図 26**）[*9]．

7.3 Accelerate

Accelerate [13] は HPC 計算のための EDSL で，Haskell のライブラリとして提供されている．並列なプログラムを Haskell のプログラムとして記述することができるので Haskell の強力な型システムをそのまま利用することができる．Accelerate にも配列型が存在しているが長さに関しては配列の形（何次元か）しか型に入っておらず，実際の長さは実行時情報として持っている．フィルター関数の返り値は結果と結果の形のペアとなる．

8. まとめ

本論文では，GPGPU プログラミングのための中間言語 Lift IL の問題として動的に長さが決まる配列を使用できないことを指摘した．その問題を解決するために Lift IL を存在型で拡張し，配列型の長さを隠蔽することを提案した．また，配列長を隠蔽，展開する操作である pack と unpack を適切な場所に自動で挿入する方法も提案した．

論文 [8] の内容に従って Lift IL コンパイラを作成し，そのうえで提案に基づいて動的長配列と，pack と unpack の自動挿入，フィルター関数を実装した．評価では動的長配列および pack と unpack の自動挿入の有用性を示すために Project Euler 問題集の最初の 30 問について検討を

[*7] このプログラムは https://github.com/prg-titech/min-lift/tree/master/pro2019-4 にある（以下同様）

[*8] このことは厳密な論証を行っていないが直感的には次のように理解できる．min-lift の型規則は存在型の導入を配列型のみに限定している．さらに配列の長さはサイズ型にしかならない性質がある．なぜならば，配列型はプログラムの入力とプリミティブ関数のみで新たに作られ，入力に関してはその性質が構文によって保証されており，プリミティブ関数に関してはその性質を満たすものしかないからである．

[*9] https://www.idris-lang.org/docs/1.2/base_doc/docs/Data.Vect.html

行った．その結果，従来の Lift IL では解法を見つけることができなかったが，今回の拡張により新たに 3 問の解法を見つけることができた．また，型システムとコード生成，pack と unpack の自動挿入が問題なく機能していることも確認した．

また，フィルター関数を含むプログラムの実行時間が妥当であることを示すためにフィルター処理を含むプログラムを Lift IL と OpenCL でそれぞれ記述し実行時間を比較した．その結果，Lift IL 版の実行時間は OpenCL の約 1.08 倍であることを確認した．

本研究で動的長配列の有用性を確認できたので，Lift IL の応用範囲を広げるために動的長配列を扱う関数の追加，if 式などの実装を今後の課題としてあげる．また，動的長配列を含む高級言語プログラムから Lift IL への変換規則の追加も将来的には行う．

参考文献

[1] Hormati, A.H., Samadi, M., Woh, M., Mudge, T. and Mahlke, S.: Sponge: Portable stream programming on graphics engines, *ASPLOS XVI*, pp.381–392 (2011).

[2] Dubach, C., Cheng, P., Rabbah, R., Bacon, D.F. and Fink, S.J.: Compiling a high-level language for GPUs: (via language support for architectures and compilers), *PLDI '12*, pp.1–12 (2012).

[3] Steuwer, M., Fensch, C., Lindley, S. and Dubach, C.: Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code, *ICFP 2015 Proc. 20th ACM SIGPLAN International Conference on Functional Programming*, pp.205–217 (2015).

[4] Pizzuti, F., Steuwer, M. and Dubach, C.: Position-dependent Arrays and Their Application for High Performance Code Generation, *Proc. 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, FHPNC 2019*, pp.14–26, ACM (online), DOI: 10.1145/3331553.3342614 (2019).

[5] Benjamin C. Pierce（著），住井英二郎（監訳），遠藤侑介（訳），酒井政裕（訳），今井敬吾（訳），黒木裕介（訳），今井宜洋（訳），才川隆文（訳），今井健男（訳）：型システム入門プログラミング言語と型の理論，オーム社 (2013).

[6] Philozoff, S.：OpenCL：ネィティブから，より洞察力のあるプログラミングへ，入手先 ⟨https://www.mql5.com/ja/articles/407⟩.

[7] 住井英二郎：K 正規化（kNormal.ml），入手先 ⟨http://esumii.github.io/min-caml/tutorial-mincaml-9.htm⟩.

[8] Steuwer, M., Remmelg, T. and Dubach, C.: Lift: A functional data-parallel IR for high-performance GPU code generation, *Proc. 2017 International Symposium on Code Generation and Optimization*, (*CGO '17*) pp.74–85 (2017).

[9] Hillis, W.D., Guy L. and Steele, J.: Data parallel algorithms, *Comm. ACM – Special Issue on Parallelism*, Vol.29, No.12, pp.1170–1183 (1986).

[10] Xi, H. and Scott, D.: Dependent Types in Practical Programming, *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages*, pp.214–227, ACM Press (1998).

[11] Xi, H. and Pfenning, F.: Dependent Types in Practical Programming, *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.214–227, Association for Computing Machinery (online), DOI: 10.1145/292540.292560 (1999).

[12] Brady, E.: Idris — A Language with Dependent Types, available from ⟨https://www.idris-lang.org/⟩.

[13] Ouwehand, C., Hijma, P. and Fokkink, W.J.: GPU Programming in Functional Languages: A Comparison of Haskell GPU Embedded Domain Specific Languages (2013).

付　　録

A.1　min-lift の構文

構文

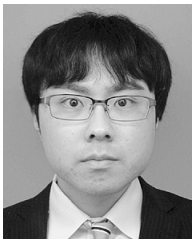
program ::=(**lift**
(i_0 ... i_m)
(U_0 ... U_n)
(**lambda** (i'_0 ... i'_n) t))

t ::= x	*variable*
n	*integer literal*
f	*float literal*
true	*constant true*
false	*constant false*
$\#n$	*constant size literal*
(**lambda** (i_0 ... i_n) t)	*abstraction*
(t_0 ... t_n)	*application*
(**let** i t t)	*let*
(**pack** t)	*pack*
(**unpack** i t_1 t_2)	*unpack*
`mapSeq` | `mapGlb` | `mapWrg`	
`mapLcl` | `reduceSeq`	
`filterSeq` | `filterGlb`	
`split` | `join` | `zip`	
`get1` | `get2` | `id`	
`toGlobal` | `toLocal`	
`toPrivate`	
$*$ | $+$ | $<$ | $>$	
`+i` | `*i` | `/i` | `=i`	
`mod` | `or` | `and` | `not`	*embedded functions*

i は識別子．

U ::=(**array-type** U i)	
float	
int	

U は型規則中に現れる型のうち，プログラムの入力として許されるものに限ったものである．



新美 和生

1995 年生．2019 年東京工業大学理学部情報科学科卒業．現在，同大学情報理工学院数理・計算科学系数理・計算科学コース修士課程に在籍．

増原 英彦　（正会員）

東京工業大学東京工業大学情報理工学院教授．1992 年東京大学理学部情報科学科卒業．1994 年同大学大学院理学系研究科修士課程修了．博士（理学）．1995 年から 2013 年まで東京大学大学院総合文化研究科で助手・講師・准教授．プログラミング言語の設計・実現方式，ソフトウェア開発環境に興味を持つ．ACM，日本ソフトウェア科学会各会員