平成31年度 学士論文

GPU向けデータ並列中間言語 Lift におけるフィルター関数の設計と実装

東京工業大学 理学部 情報科学科

学籍番号 15B16310

新美 和生

指導教員

増原 英彦 教授

平成31年1月31日

概要

LIFT プロジェクトは GPGPU プログラミングの現在の問題点である、実行速度を速くしようとするとプログラムがハードウェア依存になってしまう点を解決する。LIFT プログラムを記述する際に用いられる LIFT IR は GPGPU 向けのデータ並列な関数型 IR(Intermediate Repserentation: 中間表現) である。本研究では LIFT IR をフィルター関数を使用できるように拡張し、統計やデータベース、シミュレーションなどのより広い分野に応用できるようにすることを目指す。LIFT IR の仕様の範囲内ではフィルター関数を使用できるようにするのは不可能なので、本研究では LIFT IR の型システムに存在型を導入することによってフィルター関数の型を表現可能にし、LIFT IR 上でのフィルター関数を実現した。その上で予備評価を行い LIFT で記述したフィルター関数を含むプログラムと同じ処理を行う手で記述した OpenCL プログラムとで約 10%の差があることが示された。

謝辞

本研究を遂行するにあたり直接のご指導を頂いた、増原英彦教授、型に関する助言を頂いた、青谷知幸助教、研究やスライドに関する助言を頂いた、研究室のメンバーの皆様には心より感謝の意を表する。

目 次

第 1 章　はじめに **7**

第 2 章　背景 **10**

2.1 Lift 10

2.1.1 実行モデル 10

2.1.2 処理方法の指定 11

2.1.3 型システム 12

2.1.4 配列の実行時の表現方法 13

2.2 Prefix Sum 14

2.3 存在型 14

第 3 章　問題点 **15**

第 4 章　解決方法 **17**

4.1 フィルター関数の GPU 上での実現 17

4.2 型システムの修正 17

4.2.1 pack, unpack の自動挿入 20

4.2.2 コンパイル後の OpenCL プログラムでの存在型と pack, unpack の表現 21

第 5 章　実現 **23**

5.1 Lift コンパイラの再実装 23

5.2 Lift IR の拡張 24

5.2.1 存在型, let, unpack, pack の追加 24

5.2.2 filterSeq, filterGlb の追加 25

5.3 未実現項目 26

第 6 章　予備評価 **31**

6.1 Lift で実装したフィルター処理と手で書いたフィルター処理との比較 31

第 7 章　関連研究 **35**

7.1 Idris 35

7.2 Accelerate 35

第 8 章 まとめ 36

図 目 次

2.1 Lift の実行モデル 10
2.2 OpenCL のスレッドとメモリモデル 11

6.1 予備評価プログラムの実行時間 32

表 目 次

6.1 予備評価プログラムの実行時間 31

第1章　はじめに

本来は画像処理などに用いられるGPUを汎用(GPGPU: General-Purpose computeing on GPU)、つまり機械学習などの計算に用いる事が多くなっている。

GPGPUにはパフォーマンスの移植性の問題がある。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、高速なプログラムを書こうとするとハードウェアの特性を活用するプログラミングをすることになる。そのようなプログラムは対象のハードウェアに依存することになり、他のハードウェアで動かそうとすると実行速度が出ないことがある。

StreamIt[5]やLiquidMetal[3]は対象のハードウェアごとに専用のバックエンドを用意することでこの問題の解決を試みたが、ハードウェアの変更が生じた際にバックエンド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 プロジェクト[1]は別のアプローチで解決を試みた。Lift とはGPGPU向けのデータ並列な関数型IR(Intermediate Repserentation: 中間表現)のことである。ユーザーが記述する高級言語が書き換え規則によりLift IRに変換され、ハードウェアに最適なLift IRプログラムが探索される。そこからLift コンパイラがLift IRをOpenCLのプログラムに変換する。ハードウェアに依存した最適化を自動で探索することにより、高級言語プログラムからハードウェアへの依存を無くす[7]。

以下はユーザーが記述する高級言語の例(プログラム1.1)とそこから変換されるLift プログラム(プログラム1.2)、OpenCLプログラムの例(プログラム1.3)である[7]。

```
1  λxs. map (λx. x * 3) xs
```

プログラム 1.1: ユーザーが記述する高級言語

Lift IRにはmap(配列の写像)やreduce(配列を縮約)、split(配列を分割する)、join(配列を結合する)、slide(窓の範囲で配列を生成する)といった基本的な配列操作を行う関数が用意されており、これらを組み合わせてプログラムを構成する。

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

```
1 λ xs. (join o mapWorkgroup(
2                   joinVec o mapLocal(
3                     mapVec( λ x. x * 3)
4                   ) o splitVec 4
5                 ) o split 1024) xs
```

プログラム 1.2: 高級言語から変換され、対象のハードウェアに最適化された LIFT プログラム

```
1  int4 mul3(int4 x) { return x * 3; }
2  kernel vectorScal( global int * in,out, int len){
3    for ( int i= get_group_id ; i < len/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 } } }
```

プログラム 1.3: LIFT プログラムから変換された OpenCL コード

Lift IR に用意されていない関数の 1 つに、与えられた述語関数を満たす配列要素を集めるフィルター関数が挙げられる。これは統計やデータベース、人工知能、シミュレーションなどに用いられており [2] これらの処理は GPU と相性が良い。このフィルター関数が使えるようになると Lift プログラムの応用範囲が広がるが、Lift IR の仕様の範囲内では実現するのは不可能である。

そこで本研究では Lift の型システムを拡張することにより Lift 上でのフィルター関数を実現する。本稿では GPU 上に並列なフィルター関数を実装する方法とどのように型システムを拡張したかを説明する。

本稿の残りは、次のような構成からなっている。第 2 章は背景を述べる。第 3 章では、現状の問題点を、第 4 章では、どのようにしてフィルター関数を設計したかを、第 5 章では、どのようにしてフィルター関数を実装したかを、第 6 章では、本研究の評価を、第 7 章では、関連研究を、そして第 8 章でまとめを述べる。

第2章　背景

本研究の下地となった事柄について述べる。

2.1 Lift

第 1 章でも述べたように Lift [1]とは GPGPU 向けのデータ並列な関数型 IR のことである。Lift は純粋関数型言語で、map や reduce などの関数を合成や入れ子にすることによりプログラムを抽象的に表現することが可能となる。

2.1.1 実行モデル

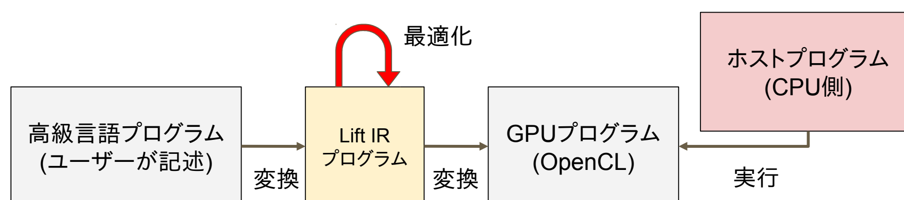

図 2.1: Lift の実行モデル

Lift の実行モデルを述べる。まずユーザーが高級言語プログラムを記述し、それを低級な Lift IR に変換する。それを GPU 上で実行される GPU プログラムに変換し、最後にホストプログラムが GPU プログラムを実行する (図 2.1)。Lift IR の段階で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 [7] ことにより最適なプログラムを生成する。高級な関数型 IR の他に Lift IR を用意している理由は Lift IR の方で複数回書き換え規則を適用していく過程でハードウェアに特化した情報を保持するためである。

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

2.1.2　処理方法の指定

Lift IR では対象の配列の処理方法や結果の格納場所を指定することができる。具体的に説明する前に OpenCL のスレッド、メモリモデルについて述べる。

OpenCL は GPU などのデバイスという大きなまとまりがありその中にワークグループが複数あり、各ワークグループ内にスレッドに相当するワークアイテムが複数ある (図 2.2)。処理するデータがある場合にそれをどのようにワークグループやワークアイテムに割り振るかはプログラマが決定する。

各まとまり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。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稿では割愛する。各まとまりはそれを包含するまとまりに対応するメモリにアクセスできる。例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。

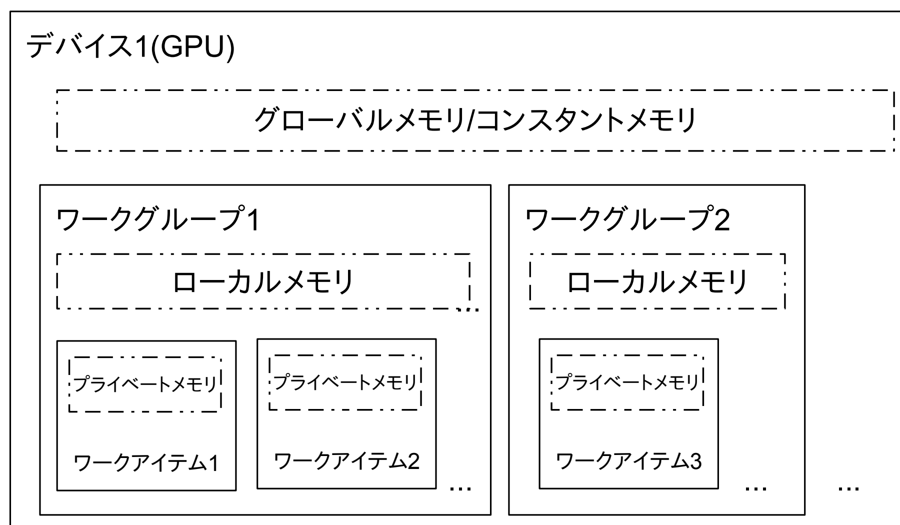

図 2.2: OpenCL のスレッドとメモリモデル

Lift IR では、要素に対してスレッドをどのように割り当てるかを指定する。例えば以下のプログラム 2.1 は `mapGlb` によってすべてのワークアイテムを利用する map(写像) 処理を行う。このプログラムに入力 `[1, 2, 3, 4, 5, 6]` を与えるとそれぞれの要素が対応するワークアイテム (`1` が Work Item1 に、`2` が Work Item2 に... という具合) に割り振られる。

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

プログラム 2.1: 入力配列 (要素の型は Float、長さは N) を 2 倍する Lift プログラム。

他に `mapSeq` や `mapWrg`、`mapLcl` というものがある。`mapSeq` は先程の入力を与えるとそれぞれのワークアイテムで入力が処理される (`[1, 2, 3, 4, 5, 6]` が Work Item1, `[1, 2, 3, 4, 5, 6]` が Work Item2... という具合)。`mapWrg` は先程の入力に対して、各ワークグループで処理される (`1` が Work Group1 に、`2` が Work Group2 に... という具合)。`mapLcl` は `mapWrg` 内に記述する必要があり、入力をワークグループ内のワークアイテムで処理する。プログラム 2.1 の `mapGlb(...)` の部分を `mapWrg(mapLcl(...), xs)` のように書き、入力 `[[1, 2], [3, 4]]` を与えると `[1, 2]` が Work Group1 で処理され、その中のワークアイテムで処理するので `1` は Work Item1, `2` は Work Item2 で処理される。`[3, 4]` についても同様である。

Lift IR でのメモリの指定は `toGlobal`, `toLocal`, `toPrivate` を用いる。それぞれグローバルメモリ、ローカルメモリ、プライベートメモリに格納することを指定する。例えば、`toGlobal(mapGlb f)(xs)` は配列 `xs` を関数 `f` で写像しその結果をグローバルメモリに格納する Lift プログラムである。

2.1.3　型システム

Lift の型システムの特徴を述べる。Lift は静的で強い型付けの言語で、コンパイル時にすべての式の型が決定している。また、コンパイラが型推論を行うので、ソースコード中には型注釈は存在しない。

Lift は配列型に依存型を導入している。依存型とは値に依存する型のことで、Lift では配列の長さも型に含まれている。例えば整数型 (`Int`) で長さ N の配列型は `Array[Int, N]` と表される。そして `N` も型なのでコンパイル時にシンボリックに大きさが等しいことが検査される。

例えば zip 関数は型が $Array[a, s] \to Array[b, s] \to Array[(a, b), s]$ で表され、これは 2 つの入力配列の長さが等しいことを要求し、結果の配列の長さも入力配列の長さに等しくなることを表している。これにより `zip(xs,`

```
(lift
 (N)
 ((array-type float N))
 (lambda (xs)
  (o
   join
   (mapSeq (lambda (ys)
    (mapSeq (lambda (y) (* y 2.0f)) ys)))
   (split #2)
   xs)))
```

プログラム 2.2: 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している。

`map(f, xs))` は型検査に合格し、`zip(xs, ys)`(`xs` は `Array[Int, M]`, `ys` は `Array[Int, N]`) は不合格となる。また、配列の長さに対して乗算や除算を行うことが出来る。例えば $split^n$ 関数は配列を n 分割する関数で、$n \to Array[a, m] \to Array[Array[a, n], m/n]$ と表すことが出来る。$split^2$ に `Array[Int, N]` 型の配列を適用した結果の型は `Array[Array[Int, 2], N/2]` となる。

2.1.4 配列の実行時の表現方法

LIFT では実行時に受け取る大きさ (以下では引数の `N`) 以外に新たに配列の大きさを表す変数を用意することはなく、コンパイル後プログラム中のすべての大きさを表す式はホストプログラムがカーネルに与える引数と定数で表すことができる。

例えば、プログラム 2.2 が OpenCL にコンパイルされるとプログラム 2.3 になる。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。プログラム 2.2 の `xs` の型は `Array[Float, N]` で、9 行目の `split` 後の型は `Array[Float, N/2]` なので長さは `N/2`、`ys` の型は `Array[Float, 2]` なので長さは 2 となる。

```
kernel void KERNEL(
    float *xs,
    float *result,
    int N) {
  ...
```

```
6     // split後のmapSeq
7     for (int i = 0; i < N / 2; i++) {
8       // ysについてのmapSeq
9       for(int j = 0; j < 2; j++) {
10        ...
11      }
12    }
13  }
```

プログラム 2.3: プログラム 2.2 のコンパイル結果

2.2 Prefix Sum

Prefix Sum とは各要素 x_i について i 番目までの和である。例えば [1, 2, 3, 4, 5] に対する Prefix Sum は [1, 3, 6, 10, 15] となる。

Prefix Sum を並列に求めるアルゴリズム [4](以下アルゴリズム A と表記) を紹介する。アルゴリズム A の疑似コード 2.4 を以下に示す。ceil は天井関数、log2 は底を 2 とした対数関数、pow(a, b) は a の b 乗 (a^b)、in parallel は for 文が並列に処理されることを表す。

```
1  function prefix-sum(in, out, n)
2    for i in 0 to ceil(log2(n)) - 1 do
3      for j in 0 to n - 1 do in parallel
4        if j < pow(2, i) then
5          out[j] = in[j]
6        else
7          out[j] = in[j] + in[j - pow(2, i)]
8      swap in and out
9    if ceil(log2(n)) % 2 == 0
10     out = in
```

プログラム 2.4: アルゴリズム A の疑似コード

2.3 存在型

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 $\{\exists X, T\}$ と表すことにする。ここで X は型変数、T は型であり内部に X の出現を許す。

第3章　問題点

現状での問題点を述べる。

第1章では LIFT の仕様の範囲内でフィルター関数を実装するのは不可能であると述べたが、その理由は LIFT で用意されている関数と型システムの制約の2つである。

第1に、フィルター関数は LIFT に用意されている関数では実装することが出来ない。LIFT に用意されている配列に対する関数で長さが変わるものは reduce(配列を単一の値に縮小)、split(配列を分割する)、join(配列を結合する)、slide(窓の範囲で配列を生成する) であるが、配列を新たに生成したり配列に要素を追加する関数がないことを考慮するとこれらの関数ではフィルター関数が実現できないことがわかる。

第2に、LIFT の型システムは第2章で述べたように配列の長さがコンパイル時に決定している必要がある、という制約により結果が可変長配列となるフィルター関数の型を表現できない。この長さの情報は配列長に関するプログラムの妥当性の検査に用いられる。例えば2つの等しい長さの配列を受け取る zip 関数があるがこの長さの検査は配列型に含まれる長さを用いる。しかし、可変長配列の場合はその長さが実行時に決定するので型検査時に可変長配列を zip のような関数に適用する式の妥当性を検査できない。

なので、フィルター関数を実装するに当たって次の要求を満たす型が必要となってくる。

- コード生成時に既存の配列型 (長さがコンパイル時に決定している) と区別したい
- 長さが不定な配列でも正しく型検査を行いたい。例を以下に挙げる。
  - `let xs' = filter(f, xs) in zip(xs', map(g, xs'))` のようなプログラムは合格させたい (filter を適用した配列に map を適用しても長さは同じとなる)。
  - `zip(filter(f, xs), filter(g, xs))` のようなプログラムは異なる filter から生成された配列はそれぞれが異なる長さとなるので不合格にさせたい。

– `let xs'= map(ys → filter(f, ys), xs) in zip(xs'[0], xs'[1])`(ただし `xs` は `Array[Array[Int, M], N]`) のようなプログラムは不合格にさせたい (`xs'[0]` と `xs'[1]` は同じ長さとは限らないので)。

第4章　解決方法

4.1　フィルター関数の GPU 上での実現

GPU 上で並列にフィルター処理を行うアルゴリズムは複数存在するが本研究では Prefix Sum を並列に処理するアルゴリズム A(2.2 を参照) を用いる。アルゴリズム A を用いてフィルター処理を行うアルゴリズムの疑似コード 4.1 を以下に示す。

```
1  // xsは入力配列、fは述語関数
2  function filter(xs, f, result)
3    bitmap = xsと同じ長さで0で初期化された配列
4
5    for i in 0 to size of xs in parallel
6      bitmap[i] = f(xs[i])
7
8    indices = xsと同じ長さで0で初期化された配列
9    prefix-sum(bitmap, indices, size of xs)
10
11   for i in 0 to size of xs in parallel
12     if bitmap[i] == 1 then
13       result[indices[i] - 1] = xs[i]
```

プログラム 4.1: 並列フィルター処理を行う疑似コード

通常、並列に処理した後はバリア同期が必要になるのだが、OpenCL ではワークグループを跨ったバリア同期が存在しないので、カーネルを複数回実行する必要がある。

4.2　型システムの修正

第 2 章で説明した通り、Lift の配列型は要素の型 I と要素数 N からなる。一方、フィルター関数の結果の配列の要素数はコンパイル時には不定である。

そこで本研究では存在型を用いて実行時に長さが決定するような配列の型を表し、長さに関する同一性を判定できるような式を制限する。本研究では配列の長さを不定としたいので取りうる存在型は $\{\exists X, Array[T, X]\}$ のみとなる。ここで X は自然数の上を動く型変数である。一般の存在型は $\{\exists X, T\}$ と表され、T はどのような型でも良い。また存在型の値を扱うのに必要となる pack と unpack を導入し、これらを自動挿入する。

存在型を導入することによって第 3 章で述べた問題をどのように解決するかを述べる。

1 つ目の存在型と他の型の区別は明らかに可能である。

2 つ目の正しい型検査は pack, unpack という規則により保証する。unpack は存在型の除去規則の適用場所を指定する。存在型 $\{\exists X, Array[T, X]\}$ である xs を具体的な型にして xs' に束縛し $body$ を評価する。例えば xs の型が $\{\exists X, Array[Int, X]\}$ の場合、xs' の型は $Array[Int, X1]$ となる。このときの $X1$ は一意であれば何でも良い。通常の unpack は変数名の他に型名 (X の部分) を取るが LIFT では配列の長さを直接扱う方法がないので省略している。

構文 (項)

$$term ::= ...$$
$$\mathbf{unpack}\ xs' \ = \ xs\ \mathbf{in}\ body$$

型付け規則

$$\frac{\Gamma \vdash t_1 : \{\exists X, Array[I, X]\} \qquad \Gamma, X, x : Array[I, X] \vdash t_2 : T_2}{\Gamma \vdash \mathbf{unpack}\ x = t_1\ \mathbf{in}\ t_2 : T_2}\ FV(T_2) \cap \{X\} = \phi$$

ただし $FV(T)$ は T の自由変数である。

上記の規則から解るように unpack した配列型の長さは $body$ の自由変数に出現してはいけないので unpack した配列は pack する必要がある。もちろん、reduce など関数で配列の長さを無くしてしまえば pack する必要は無くなる。

一方、pack は存在型の導入規則を適用する場所を指定する。例えば xs の型が $Array[Int, N]$ の場合、**pack** xs の型は $\{\exists X, Array[Int, X]\}$ となる。これも通常、どのような存在型になるか ($\{\exists X, T\}$ の X と T) を取るが明らかなので省略している。

構文 (項)

$$term ::= \ldots$$
$$\quad \mathbf{pack}\ xs$$

型付け規則

$$\frac{\Gamma \vdash t_2 : [X \mapsto N]Array[I, X]}{\Gamma \vdash \mathbf{pack}\ t_2 : \{\exists X, Array[I, X]\}}$$

これらを踏まえてフィルター関数の型は以下のように表される。

フィルター関数の型

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

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掲載する (プログラム 4.2)。

```
1 (lift
2  (N)
3  ((array-type float N))
4  (lambda (xs)
5    (unpack ys (filterSeq (lambda (x) (< x 0.5f))
       xs)
6     (pack (mapSeq (lambda (x) (* x 2.0f)) ys)))))
```

プログラム 4.2: 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)

このプログラムの型付けを行うと `filterSeq` の結果の型は $\{\exists X, Array[Float, X]\}$ となり、それを `unpack` することにより `ys` の型は例えば $Array[Float, X1]$ となる。`ys` に対して `mapSeq` を行っても型は変わらず、その後 `unpack` を行うと型は $\{\exists X, Array[Float, X]\}$ となる。プログラム 4.2 の導出木を以下に示す。

$$\frac{\Gamma \vdash \texttt{(filterSeq (...) xs)} : \{\exists X, Array[Float, X]\} \qquad T}{\Gamma \vdash \texttt{(unpack ys (filterSeq (lambda (x) (< x 0.5f)) xs) (pack (mapSeq (lambda (x) (* x 2.0f)) ys)))} : \{\exists X, Array[Float, X]\}}$$

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

$$\text{ただし}\Gamma = xs : Array[Float, N]$$

3 章で挙げたプログラム片についても型付けを行う。以下 `xs` の型を $Array[I, N]$ とする。

- `let xs’ = filter(f, xs) in zip(xs’, map(g, xs’))` は pack と unpack を用いて `unpack xs’ = filter(f, xs) in pack(zip(xs’, map(g, xs’)))` と書き換える必要がある。`xs’` の型は例えば $Array[I, X_1]$ となり、`g` が恒等写像だとすると `map(g, xs’)` の型は $Array[I, X_1]$ になる。`xs’` と `map(g, xs’)` の長さは同じ X_1 なので zip に適用しても問題はない。

- `zip(filter(f, xs), filter(g, xs))` も同様にプログラム 4.3 と書き換える必要がある。これを型付けしようとすると、`xs’` は例えば $Array[I, X_1]$、`xs’’` は $Array[I, X_2]$ となる。`xs’` と `xs’’` の長さは異なるのでこれらを zip 関数に適用した式は型検査に不合格となる。

- `let xs’= map(ys → filter(f, ys), xss) in zip(xs’[0], xs’[1])` も同様にプログラム 4.4 と書き換える必要がある。`xss` の型を $Array[Array[I, M], N]$ としてこの式を型付けしようとすると、`xss’` は $Array[\{\exists X, Array[I, X]\}, N]$ となり、`xs’` と `xs’’` は例えば $Array[I, X_1]$ と $Array[I, X_2]$ となり、上記と同様に長さが異なるので zip 関数に適用すると不合格となる。

```
1 unpack xs’ = filter(f, xs) in
2   unpack xs’’ = filter(g, xs) in
3     zip(xs’, xs’’)
```

プログラム 4.3: 異なる関数から抽出された配列を zip 関数に適用する式

```
1 let xss’ = map(ys → filter(f, ys), xss) in
2   unpack xs’ = xss’[0] in
3     unpack xs’’ = xss’[1] in
4       zip(xs’, xs’’)
```

プログラム 4.4: 可変長配列が要素となっている配列の異なる 2 要素を zip 関数に適用する式

4.2.1　pack, unpack の自動挿入

存在型の値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

うにしたいが 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。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 [9]。例えば `map(g, filter(f, xs))` という式は `let xs2 = filter(f, xs) in map(g, xs2)` という式に変換される。

2. 以下の書き換え規則により let を pack, unpack を含む式に置き換える。

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

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

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

4.2.2 コンパイル後の OpenCL プログラムでの存在型と pack, unpack の表現

Lift IR 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存在型を、そのデータと長さを含んだ構造体で表現する。既存の配列がコンパイル後プログラムでどのように表現されるかは、すでに 2.1.4 で説明した。ここに存在型を導入すると、配列の長さがコンパイル時に定まらなくなるので不都合が起こる。なので存在型の場合は新たに配列の長さを表す変数を用意する必要がでてくる。フィルターを実行するプログラムは複数のカーネルを跨ぐ可能性があるのでそれを考慮する必要がある。例えば、`let x = filter(f, xs)` *in let y = filter(f, xs)* `in map(g, x)` のようなプログラムでは斜体のコード間で配列 x の長さをやり取りする必要がある。そこで、存在型の配列はそのデータと長さを含んだ構造体で表現することにする。例えば $\{\exists X, Array[Int, X]\}$ はプログラム 4.5 のように表現される。構造体にまとめることによってカーネル間の配列のやり取りが容易となる。

```
1 struct existential_int {
2   unsigned length;
3   int data[];
```

```
4 }
```

プログラム 4.5: コンパイル後の存在型の配列の例

pack と unpack は型検査時にしか意味を持たずコード生成時には不要な操作となるので OpenCL プログラム内では特別な操作は行われない。

第5章　実現

本研究に関する実装について述べる。まず、Lift IR の拡張を行うに先立ち Lift コンパイラの再実装を行い、実装した Lift コンパイラに存在型、let、pack、unpack を実装し、filterSeq と filterGlb を実装した。最後に未実装項目について述べる。

5.1　Lift コンパイラの再実装

Lift IR の拡張を行うに先立ち、後々の拡張性を考え Lift コンパイラの論文 [8] を参考に、Lift コンパイラの再実装から行った。実装した処理の流れの概略を説明する。

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. 後の処理を楽にするためのプログラムの正規化。行うことは以下の 2 つである。

    - 関数合成の書き換え。`(o f g h)` のような式を関数合成演算子 `o` を含まない式 `(f (g h))` に変換する。
    - 部分適用の消去。`(((f x) y) z)` のような式を `(f x y z)` に変換する。

3. 型検査と型推論。すべての式は型を持つのでその型に矛盾がないかを検査する。また、Lift は式に型を記述しないので各式の型を推論する。これらは型再構築により同時に実現することが出来る。「型システム入門 プログラミング言語と型の理論 22 章 型再構築」[10] を参考に、与えられた式から制約を生成し、その後 unify するアルゴリズムを実装した。

4. 配列へのアクセスの解決。Lift 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。まず、View Construction というある式がメモリにどのようにアク

セスしているか、という情報 (View) を生成し、View Consumption で実際の添字を生成する (実際に生成するのはコード生成時)。

5. 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出来る。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求める。

6. コード生成。ここまでに求めた情報 (型、View、メモリ領域) を元に、OpenCL のコードを生成する。

上記に加え、Lift IR を拡張するための準備として Boolean 型、論理演算子の追加を行った。

5.2　Lift IR の拡張

次に、本研究の目的であるフィルター関数を実現するために行った実装について述べる。

5.2.1　存在型, let, unpack, pack の追加

存在型

まず、型を示すクラス (`Type`) に `Type(ty)` のフィールドを持つ、存在型 `Existential` を追加する。次に unify する規則と型変数を置換する処理を追加する。これらは他の型と同様の処理なので詳細は割愛する。最後にコード生成器を存在型に対応させる。具体的には、存在型を処理する際にその長さの変数を利用する、フィルター処理の後に長さの変数に設定するということをしている。

let, unpack

let と unpack は存在型の処理以外は同じなのでまとめて述べる。まず、let と unpack は関数ではなく構文なのでそれを表すクラス (プログラム 5.1) を追加し、構文解析器を修正して let と unpack を含んだプログラムが正常に構文解析をできるようにする。次に、型検査器の制約を生成する処理に let と unpack の場合を追加する。具体的な処理を以下に述べる。

1. `value` の型検査を行う。

2. `unpack` が偽の場合 (let)、`id` と `value` の型の組を新たな環境に追加し、`body` の型検査を行う。

   `unpack` が真の場合 (unpack)

   (a) 一意の長さ型を生成し、そこから存在型 `existType` を生成する。

   (b) `id` と `existType` の組を新たな環境に追加し、`body` の型検査を行う。

   (c) `valueType` と `existType` の組で制約を生成する。

最後に、コード生成器の処理に let と unpack の場合を追加する。具体的には `value` を評価し `id` に設定し、`body` を評価するコードを生成している。let と unpack で処理内容は変わらない。

```
1  case class Let(
2      // 識別子(letで定義する変数名)
3      val id: Identifier,
4      // letで定義する変数に設定される式
5      val value: Expression,
6      // 変数を定義した後に評価される式
7      val body: Expression,
8      // 真の場合unpack、偽の場合let
9      val unpack: Boolean
10     ) extends Expression {
11   def accept[A, R](visitor: ExpressionVisitor[A,
        R], arg: A): R = {
12     visitor.visit(this, arg)
13   }
14 }
```

プログラム 5.1: Let 式のクラス定義

5.2.2 filterSeq, filterGlb の追加

2.1.2 で説明したように Lift では配列をどのように処理するかを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ごとに処理をする filterSeq とすべてのワークアイテムを使用する filterGlb を実装した。まずは 4.2 のフィルター関数の型定義を型検査器のプログラムに追加した (プログラム 5.2)。

次にコード生成器の修正を行った。filterSeq は他スレッドとの協調を考慮する必要がないので既存の mapSeq などと同じようにコードを出力でき

```
1 ("filterSeq" -> TypeScheme(List(a, b, c), (a ->:
    Boolean) ->: Array(a, b) ->: Existential(c,
    Array(a, c)))),
2 ("filterGlb" -> TypeScheme(List(a, b, c), (a ->:
    Boolean) ->: Array(a, b) ->: Existential(c,
    Array(a, c)))),
```

プログラム 5.2: filterSeq、filterGlb の型定義 (型検査器のプログラム片)

る。filterGlb は同期の関係上カーネルを跨ぐので filterSeq のようには出力できない。そこで、一旦 filterSeq のように出力をする。この際にカーネルを分割する印を付けておき、プログラムをすべて出力してから印で分割する。

最後に作成したプログラムが動作することを示す結果を掲載する (プログラム 6.1 とそれを型付けした結果 5.3、コンパイルした出力 5.4、実行結果 5.5)。

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挙げる。

- 存在型の 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。また、4.2.2 で挙げたようなカーネル間での配列のやり取りも実現できていない。
- pack の実装。
- pack, unpack の自動挿入。

```
(lift
  (N)
  (Array(Float,N))
  (lambda
    (xs:Array(Float,N)@Some(GlobalMemory))
    (toGlobal:(Array(Float,l14) => Array(Float,
       l14))@None

    (let Identifier(ys,false)
      (toGlobal:(Existential(Array(Float,l14)) =>
          Existential(Array(Float,l14)))@None
```

```
        (filterGlb:((Float => Boolean) => (Array(
           Float,N) => Existential(Array(Float,
           l14))))@None

          (lambda
            (x:Float@None)
            (>:(Float => (Float => Boolean))@None
              x:Float@Some(GlobalMemory)
              0.5:Float@Some(PrivateMemory)):
                 Boolean@Some(GlobalMemory)):(
                 Float => Boolean)@Some(
                 GlobalMemory)
          xs:Array(Float,N)@Some(GlobalMemory)):
             Existential(Array(Float,l14))@Some(
             GlobalMemory)):Existential(Array(
             Float,l14))@Some(GlobalMemory)

        (mapGlb:((Float => Float) => (Array(Float
           ,l14) => Array(Float,l14)))@None

          (lambda
            (x:Float@None)
            (*:(Float => (Float => Float))@None
              x:Float@Some(GlobalMemory)
              2.0:Float@Some(PrivateMemory)):
                 Float@Some(GlobalMemory)):(Float
                  => Float)@Some(GlobalMemory)
          ys:Array(Float,l14)@Some(GlobalMemory
             )):Array(Float,l14)@Some(
             GlobalMemory)):Array(Float,l14)@Some
             (GlobalMemory)
           ):Array(Float,l14)@Some(GlobalMemory
              )):(Array(Float,N) => Array(Float,
              l14))@None)
```

出力 5.3: プログラム 6.1 の型付け結果

```
1 // {"ChunkSize":5,"InputSize":1,"KernelCount":3}
```

```
2
3  kernel void KERNEL(
4      const global float* restrict xs,
5      global float* result,
6      global int* result_size,
7      global int* bitmap,
8      int N) {
9    int i1 = get_global_id(0);
10
11   float x = xs[i1];
12   private bool temp2;
13   private float temp1;
14   temp1 = 0.5f;
15   temp2 = x > temp1;
16
17   bitmap[i1] = temp2;
18 }
19
20 kernel void KERNEL2(
21     const global float* restrict xs,
22     global float* result,
23     global int* result_size,
24     global int* bitmap,
25     global int* indices,
26     int N) {
27
28   int ary_size = 0;
29
30   int i1 = get_global_id(0);
31   if (bitmap[i1]) {
32     result[indices[i1] - 1] = xs[i1];
33   }
34   int l14 = indices[N - 1];
35
36   *result_size = l14;
37 }
38
39 kernel void KERNEL3(
```

```
40      const global float* restrict xs,
41      global float* result,
42      global int* result_size,
43      int N) {
44    int ary_size = 0;
45
46    global float* ys = result;
47    {
48      int i2 = get_global_id(0);
49      float x = ys[i2];
50      private float temp4;
51      private float temp3;
52      temp3 = 2.0f;
53      temp4 = x * temp3;
54
55      result[i2] = temp4;
56    }
57  }
58
59  kernel void prefix_sum(global int *xs, global int
      *ys, int i) {
60    global int *in = NULL, *out = NULL;
61    if (i % 2 == 0) {
62      in = xs; out = ys;
63    }
64    else {
65      in = ys; out = xs;
66    }
67
68    int j = get_global_id(0);
69
70    int powiof2 = pown(2.0f, i);
71    if (j < powiof2) {
72      out[j] = in[j];
73    }
74    else {
75      out[j] = in[j] + in[j - powiof2];
76    }
```

```
77 }
```

プログラム 5.4: プログラム 6.1 のコンパイル結果

```
$ ./runtime -f filter-gt-0.5-twice-lift.cl -d min
   -data
Using device: Intel(R) Gen9 HD Graphics NEO
Chunk size: 5
input data
  0: 0.185677 0.558394 0.0677897 0.161399
     0.00811924 0.989952 0.916822 0.732726
     0.722605 0.0130553
execution time: 369.861 us
execution time: 361.607 us
result size: 5
1.11679
1.9799
1.83364
1.46545
1.44521
0
0
0
0
0
execution time: 361.607 us
```

出力 5.5: プログラム 6.1 の実行結果

第6章　予備評価

本研究について予備評価を行った。評価に用いた環境を以下に述べる。

- NVIDIA TITAN Xp
- Intel(R) Core(TM) i7-5960X
- 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)
- CUDA 9.1.84
- GPU ドライバーバージョン: 390.87

6.1　Lift で実装したフィルター処理と手で書いたフィルター処理との比較

長さ 10000 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 (プログラム 6.1)、OpenCL の並列 (プログラム 6.2)、OpenCL の逐次 (プログラム 6.3) で実装しそれぞれを 1000 回実行し、カーネルの実行時間の平均を取った (表 6.1, 図 6.1)。この結果から分かることは逐次版は他の 2 つより約 8.2 倍の時間がかかっており、Lift 版は並列版の約 1.05 倍ということである。つまり、Lift 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。

種類	実行時間 (μ s)
Lift	71.2669
OpenCL の並列	67.8343
OpenCL の逐次	590.776

表 6.1: 予備評価プログラムの実行時間

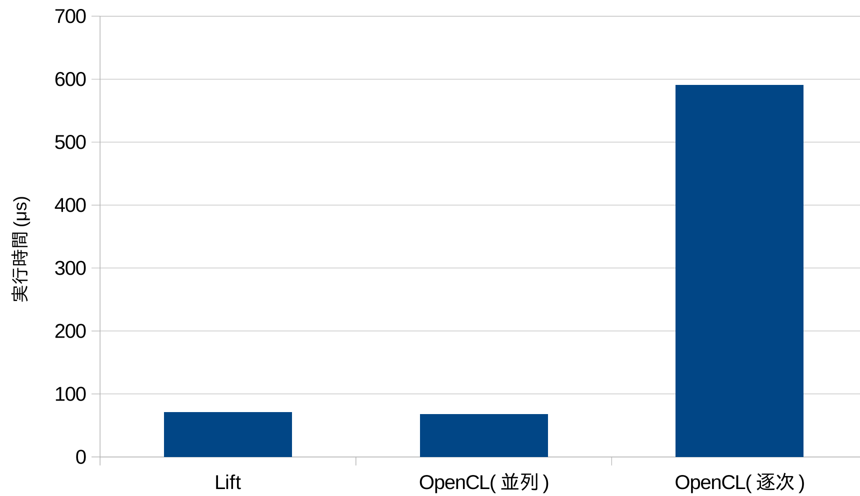

図 6.1: 予備評価プログラムの実行時間

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

プログラム 6.1: Lift バージョン (filter-gt-0.5-twice-lift.lisp)

```
1  // {"ChunkSize":5,"InputSize":1,"KernelCount":2}
2
3  kernel void KERNEL(
4    const global float* restrict xs,
5    global float* result,
6    global int* result_size,
7    global int* bitmap,
8    int N) {
9      int i1 = get_global_id(0);
10
11     float x = xs[i1];
12
```

```
13      float temp1 = 0.5f;
14      bitmap[i1] = x > temp1;
15  }
16
17  kernel void KERNEL2(
18    const global float* restrict xs,
19    global float* result,
20    global int* result_size,
21    global int* bitmap,
22    global int* indices,
23    int N) {
24
25      int i1 = get_global_id(0);
26      if (bitmap[i1]) {
27          result[indices[i1] - 1] = xs[i1] * 2.0f;
28      }
29      int l14 = indices[N - 1];
30
31      *result_size = l14;
32  }
33
34  kernel void prefix_sum(global int *xs, global int
        *ys, int i) {
35    global int *in = NULL, *out = NULL;
36    if (i % 2 == 0) {
37      in = xs; out = ys;
38    }
39    else {
40      in = ys; out = xs;
41    }
42
43    int j = get_global_id(0);
44
45    int powiof2 = pown(2.0f, i);
46    if (j < powiof2) {
47      out[j] = in[j];
48    }
49    else {
```

```
50        out[j] = in[j] + in[j - powiof2];
51    }
52  }
```

プログラム 6.2: OpenCL(並列) バージョン (filter-gt-0.5-twice-hand.lisp)

```
1   // {"ChunkSize":5,"InputSize":1,"KernelCount":1}
2
3   kernel void KERNEL(
4     const global float* restrict xs,
5     global float* result,
6     global int* result_size,
7     int N) {
8
9
10      int idx = 0;
11      for (int i = 0; i < N; i++) {
12          if (xs[i] > 0.5f) {
13              result[idx++] = xs[i] * 2.0f;
14          }
15      }
16
17      *result_size = idx;
18  }
```

プログラム 6.3: OpenCL(逐次) バージョン (filter-gt-0.5-twice-seq.lisp)

第7章　関連研究

7.1 Idris

プログラミング言語 Idris[1] 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 (プログラム 7.1)[1]。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 (プログラム 7.2)[2]。

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

プログラム 7.1: Idris の Vect の定義

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

プログラム 7.2: Idris の filter の定義

7.2 Accelerate

Accelerate[6] 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出来るので Haskell の強力な型システムをそのまま利用することが出来る。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。

[2]https://www.idris-lang.org/docs/1.2/base_doc/docs/Data.Vect.html#Data.Vect.Vect
[2]https://www.idris-lang.org/docs/1.2/base_doc/docs/Data.Vect.html#Data.Vect.filter

第8章　まとめ

本研究では GPU 向けデータ並列中間言語 Lift の応用範囲を広げるためにフィルター関数の導入を行った。この際に Lift で用意されている関数と型システムの制約により Lift の仕様の範囲内ではフィルター関数の実現は不可能だという問題がある。これを解決するために存在型とそれに伴って必要となる pack と unpack の導入、そして利便性向上の為 pack と unpack の自動挿入を Lift コンパイラのフローに加えた。これらによりフィルター関数の型が表現可能となる。

実現ではまず Lift コンパイラを再実装しその上に存在型と pack, unpack, フィルター関数を実装した。

最後に、実装したコンパイラを用いてフィルター関数の予備評価を行った。具体的には入力配列の各要素に対して 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 と OpenCL で実装し、これらの実行時間を比較した。この結果からフィルター関数を含む Lift プログラムの実行時間は同じ処理を行う手で記述した OpenCL プログラムの実行時間の約 1.05 倍であることがわかった。

今後の課題は 5.3 で述べた未実装項目の実装と予備評価に用いたプログラムより大規模なプログラムを動作可能にする、高級言語プログラムから Lift IR への変換規則の追加が挙げられる。

参考文献

[1] : Idris — A Language with Dependent Types, `https://www.idris-lang.org/`.

[2] Bakunas-Milanowski, D., Rego, V., Sang, J. and Yu, C.: A Fast Parallel Selection Algorithm on GPUs, *2015 International Conference on Computational Science and Computational Intelligence (CSCI)* (2015).

[3] 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).

[4] Hillis, W. D. and Guy L. Steele, J.: Data parallel algorithms, *Communications of the ACM - Special issue on parallelism*, Vol. 29, No. 12, pp. 1170–1183 (1986).

[5] 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).

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

[7] 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 Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pp. 205–217 (2015).

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

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

[10] 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).