より清浄なStream Fusion

小林 友明　Oleg Kiselyov

昨今，データを中心としたプログラミングパラダイムの影響力が高まっている．ストリーム処理はその一例であり，高水準かつ高性能なストリームライブラリを実現することは重要な課題である．strymonas はステージ化によってこの課題に取り組むライブラリであり，map，filter，fold 等のコンビネータからなるパイプラインに stream fusion などの最適化を施し，命令型の高効率なストリーム処理のコードを生成する．特に，これによって生成されるコードが手動で最適に記述されるものと遜色ないことは注目に値する．しかしながら，strymonas には，コード生成に用いる中間言語の汎用性や拡張性が低いことと，flat_map 後に zip するパイプラインに対して例外的に低品質のコードを生成してしまうことの問題があった．本研究ではこれらの問題を解決し，高水準かつ高性能なストリームライブラリの基盤を再建した．

1　はじめに

昨今の情報処理では人工知能や IoT のブームに起因して，データを中心としたプログラミングパラダイムが大きな役割を担っている．ストリーム処理はそのようなパラダイムの 1 つであり，ストリームという有限長あるいは無限長の系列データを逐次的に処理することを意味する [1]．逐次的な処理であるストリーム処理は万能ではないが，該当する場合には低レイテンシでワーキングメモリが定数サイズ (コンスタントメモリ) の効率的な処理が期待でき，これらの特性から組込みシステムや IoT などでよく利用されている．

その汎用性や実用性の高さから，ストリーム処理のライブラリは様々なプログラミング言語で提供されており，その多くは命令型か関数型の API を持つ．例えば命令型のストリームライブラリとしては，C の FILE や C++の input_iterator などを挙げられる．このようなライブラリによる処理は命令的で効率的だが，処理が複雑な場合の記述や読解が困難という問題をもつ．その一方で，Haskell の Data.List や OCaml の Seq のような関数型のストリームライブラリも存在する．このようなライブラリによる処理は，map，filter，fold(reduce) などの演算子 (コンビネータ) を組み合わせて記述される．そのため，処理が複雑になっても記述や読解は比較的容易なままであるが，サンクを代表とした抽象化のオーバーヘッドを多分に含んでしまうため，一般に命令型のストリームライブラリよりも効率が悪いという問題をもつ．

このようなジレンマの解決方法として，関数型のストリームライブラリに stream fusion を施すことが効果的であると知られている [1]．広義の stream fusion は，中間データ構造を使わないような計算にストリーム処理を変換する最適化であり [8]，Haskell の stream-fusion [4] [3]，OCaml の streaming [5]，Java SE 8 の Streams API(Java 8 streams) などで利用されている．しかしながら，このような stream fusion を備えたライブラリでも，タプルやオプション型の値といった入力データによらずサイズが固定の中間データ構造やクロージャによるオーバーヘッドは含むため，依然として効率に改善の余地がある．strymonas [7] [8] は，ステージ化を用いた部分評価でこの問題に取り組むストリームライブラリであり，zip や flat_map(concatMap) などの高機能な演算子を API に含みながらも殆どのパ

Highly Purifying Stream Fusion.

Tomoaki Kobayashi, Oleg Kiselyov, 東北大学, Tohoku University.

イプラインであらゆる中間データ構造やクロージャなしのストリーム処理ができるように stream fusion を行い，人間が効率的になるよう慎重に記述した命令型ストリーム処理と同等の性能を達成する．その原理は，関数型ストリーム処理のパイプラインを部分評価して最適化されたストリーム処理のコードを生成することにあり，生成されるコードがコンパイラによらず良い効率をもつことや安全であることは静的に保証される．

このように strymonas は真に高水準かつ高性能なストリームライブラリと呼ぶに相応しいものの，次の問題を抱えていた．

1. コード生成に用いる中間言語の汎用性や拡張性が低い
2. flat_map されたストリームの対に zip を行うパイプラインから生成されるコードが，以下の 2 つを含んでしまう
   2.1. 固定サイズの中間データ構造
   2.2. クロージャ

問題 1. は，より高度な最適化や新しい規則に基づいたコード生成などの機能追加を困難にする．問題 2. は，該当する場合の生成コードに対して例外的に box 化・自動メモリ管理 (GC)，クロージャの生成・呼び出しなどからなるオーバーヘッドを付加したり，低水準な言語を対象としたコード生成を困難にしたりする．なお，ストリーム処理で flat_map が使用される場面はそれほど多くないように思えるが，strymonas の filter は flat_map によって表現されていたため，

```
zip (stream1 |> filter pred1)
    (stream2 |> filter pred2)
|> ...
...
|> fold f z
```

のような実用性の高いと考えられるパイプラインも，この対象となってしまっていた．

そこで，我々はこれらの問題を解決するのために，それぞれに対して次のような手段をとった．

1. ストリーム処理の基本構造を再確認し，中間言語を再設計 (第 4. 1 節)
2. filter を flat_map の表現から分離し (第 4. 1 節)，
   2.1. 要求される固定サイズの中間データ構造を他のもので代替 (第 4. 2 節)
   2.2. コードを破壊しない安全なクロージャ変換とクロージャ周りの関数をインライン展開する正規化のフェーズを追加 (第 4. 3 節)

本研究の貢献は，strymonas の抱えていた上述の問題を系統的に解決し，以下の事項を実現したことである．

- zip と flat_map を含んだ複雑なパイプラインでも生成されるコードは型安全かつ手書きの命令的なコードと同等の性能を発揮し，効率の良い OCaml のストリームライブラリ streaming[5] の約 5 - 25 倍の性能をベンチマークで達成
- API で表現可能な任意のパイプラインをあらゆる中間データ構造なしに実現: 省メモリ・コンスタントメモリ・GC 未要求なストリーム処理
- 元の strymonas の API を拡張: map_accum, drop，drop_while，etc.
- 中間言語や最適化機能をコード生成機能から分離: OCaml や Scala だけではなく，C や LLVM などのコード生成も視野に

これ以降，本研究の問題解決による改善がなされる前の strymonas を strymonas v1，なされた後の strymonas を strymonas v2 と呼称する．

本論文の構成は以下のとおりである．第 2 節では，strymonas の概要と基本的な使用方法，また応用的なストリーム処理の記述例について解説する．第 3 節では，本研究の改善によってもたらされた一部のストリーム処理に対する strymonas の生成コードの変化と，種々のベンチマークをとった結果を示す．第 4 節では，strymonas v2 の実装について，中間言語の洗練化，残留した固定サイズの中間データ構造の除去，クロージャの除去の観点から説明する．第 5 節では，関連研究を紹介する．第 6 節では，本研究のまとめと今後の課題を議論する．

2 strymonas の概要

本節では，strymonas の概要とその使い方を説明する．strymonas の実体は，BER MetaOCaml[6] や LMS(Lightweight Modular Staging[12]) を利用した Scala などのメタ言語にライブラリとして埋め込まれ

```
1  type 'a stream
2  type +'a cde (* 抽象型 *)
3  type 'a emit = ('a → unit cde) → unit cde
4
5  (* 生産者 *)
6  val of_arr  : 'a array cde → 'a stream
7  val iota    : int cde      → int stream
8  val from_to : ?step:int → int cde → int cde → int stream
9  val unfold  : ('z cde → unit cde → ('a cde) emit) → 'z cde → 'a stream
10
11 (* 変換者 *)
12 val map         : ('a cde → 'b cde)    → 'a stream → 'b stream
13 val flat_map    : ('a cde → 'b stream) → 'a stream → 'b stream
14 val filter      : ('a cde → bool cde)  → 'a stream → 'a stream
15 val take        : int cde → 'a stream  → 'a stream
16 val map_accum   : ('z cde →  'a cde →
17                    ('z cde → 'b cde → unit cde) → unit cde) →
18                   'z cde → 'a stream → 'b stream
19 val drop        : int cde → 'a stream → 'a stream
20 val drop_while  : ('a cde → bool cde) → 'a stream → 'a stream
21 val zip_with    : ('a cde → 'b cde → 'c cde) →
22                   ('a stream → 'b stream → 'c stream)
23
24 (* 消費者 *)
25 val fold    : ('z cde → 'a cde → 'z cde) → 'z cde → 'a stream → 'z cde
26 val iter    : ('a cde → unit cde)                  → 'a stream → unit cde
```

図 1　strymonas のユーザー用高水準 API

た DSL(Embedded DSL，EDSL) である．元々採用されていたメタ言語は上述の 2 言語のみであったが，現在 Idris，Agda，C#などに広がってきている．

strymonas v2 の実装が完了しているのは本論文執筆時点で BER MetaOCaml によるもののみであり，本論文では簡単のため，これ以降特に断らない限りメタ言語を BER MetaOCaml，対象言語を OCaml とした，(BER Meta)OCaml 版 strymonas について説明する．なお，メタ言語が BER MetaOCaml の strymonas で生成対象となるのは基本的に OCaml のコードであるが，C のような低水準の言語を対象としたコード生成にも対応し始めている．

2.1　基本

図 1 に，strymonas の API を示す．strymonas が API で提供する演算子は一般の高水準ストリームライブラリと同様に生産者，変換者，消費者の 3 種類に分類される．

生産者とは，引数の値を用いてストリームを生産する関数のことであり，パイプラインの先頭で用いられる．例えば，引数で指定された配列に対してその配列のストリームを生産する of_arr(6 行目) や，ストリームの生産で汎用的に用いられる unfold(9 行目) などが該当する．

変換者とは，受け取ったストリームを異なるストリームに変換する関数のことであり，パイプラインの中間で用いられる．これには，引数で与えられる関数を使ってストリームの各要素から計算された値を新たなストリームの要素とするよう変換を行う map(12 行目) や，ストリームから指定条件を満たす要素のみを抽出する filter(14 行目) などが当てはまる．

消費者とは，与えられたストリームを処理して結果の値を返す関数のことであり，パイプラインの末尾で用いられる．特に，strymonas の消費者は，結果の値自体ではなく結果の値を計算するコードを返す．これには，ストリームの各要素を用いた反復的な計算のコードを生成する iter(foreach，26 行目) や，逐次処理のコード生成に対して汎用的に用いられる fold(25 行目) などがある．

変換者や消費者の多くは高階関数であり，そのよう

```
1  val letl : 'a cde → (('a cde → 'w cde) → 'w cde) (* 局所let *)
2  val glet : 'a cde → 'a cde                       (* 大域let(let 挿入) *)
3  val seq  : unit cde → 'a cde → 'a cde
4  val unit : unit cde
5  val cond : bool cde → 'a cde → 'a cde → 'a cde   (* if−then−else *)
6
7  (* Booleans *)
8  val bool : bool → bool cde
9  val not  : bool cde → bool cde
10 val (&&) : bool cde → bool cde → bool cde
11 val (||) : bool cde → bool cde → bool cde
12
13 (* Integers *)
14 val int   : int → int cde
15 val imin  : int cde → int cde → int cde
16 val (mod) : int cde → int cde → int cde
17 val (+)  : int cde → int cde → int cde
18 val (−)  : int cde → int cde → int cde
19 ...
20 val (<)  : int cde → int cde → bool cde
21 val (>)  : int cde → int cde → bool cde
22 val (=)  : int cde → int cde → bool cde
23 val (≤)  : int cde → int cde → bool cde
24 val (≥)  : int cde → int cde → bool cde
25 ...
26
27 (* Floating points *)
28 val float    : float → float cde
29 val truncate : float cde → int cde
30 val (+.)    : float cde → float cde → float cde
31 val (−.)    : float cde → float cde → float cde
32 ...
33
34 (* Others *)
35 val pair : 'a cde → 'b cde → ('a * 'b) cde
36 val int_array   :   int array → int array cde
37 val float_array : float array → float array cde
38 ...
```

図 2　演算子の引数記述用 EDSL

な場合にはユーザーがストリームの変換・消費方法を引数の関数で指定することになる．この引数の関数は，ストリームの要素が取り出される毎にそれを引数として発火される処理を表すため，Yacc などのパーサージェネレータに倣いセマンティックアクションと呼ぶことにする †1.

ユーザーが図 1 の演算子を組み合わせてパイプラインを作ると，strymonas はそれを評価することで stream fusion を代表とする種々の最適化が施されたコードを生成し，生成されたコードを処理系に渡すことで実際のストリーム処理が実行される．パイプラインを構築する演算子の引数には，図 1 の 2 行目記載の cde 型の値やその上の演算を与えることになる．cde 型は対象言語のコードを表現する抽象型であり，この型の値やその上の汎用的な演算は，図 2 に示す専用の EDSL を用いて記述される．次に，図 1 の API と図 2 の EDSL を利用した，基本的なパイプラインの記述例と生成コードを示す.

iota を用いて 1 以上の整数からなる無限ストリームを生産し，map によってその各要素が 2 倍された正の偶数のストリームに変換し，さらに take を用い

†1　パーサージェネレータにおけるセマンティックアクションは，定義された構文の規則に対応して構文解析時に呼び出される関数のことである．

て始めの 10 個の要素のみからなるストリームに変換し，最後に fold によって要素を加算して消費するパイプラインは，

```
iota (int 1)
|> map (fun e → int 2 * e)
|> take (int 10)
|> fold (+) (int 0)
```

と記述できる．このパイプラインは演算子の引数が図 2 の EDSL で記述されているため，通常の OCaml のストリームライブラリによって記述されているパイプラインと見かけ上ほとんど区別がつかない．次に，このパイプラインから生成されるコードを示す．

```
let lv_1 = ref 0 in
let lv_2 = ref 1 in
let lv_3 = ref 0 in
while (! lv_3) ≤ 9 do
  incr lv_3;
  let lv_4 = ! lv_2 in
  incr lv_2;
  let lv_5 = 2 * lv_4 in
  lv_1 := ((! lv_1) + lv_5)
done;
! lv_1
```

この生成コードは抽象化によるオーバーヘッドがあるような処理を含まず，手書きで高効率になるように記述されたストリーム処理と遜色ない．

以上のように演算子の引数を図 2 の EDSL を利用して記述すると，ユーザーに BER MetaOCaml 特有のステージ化注釈の存在を隠蔽したままコード生成ができるようになったり，生成コードに部分静的なデータを利用した高度な最適化をかけられるようになるといった恩恵を受けられる．ただし，外部ライブラリを利用した OCaml 特有の処理を行うコードを生成する場合などには，図 2 の EDSL は BER MetaOCaml のステージ化注釈を完全に隠蔽することはできない．第 2.2 節ではそのような場合のパイプラインの記述について記述する．

2.2 応用

本節では，前節で紹介した strymonas の機能を用いて，より実用的なストリーム処理を行うパイプラインの記述方法について説明する．ただし，OCaml のライブラリを利用した処理のコードを生成するパイプラインを簡潔に記述するため，図 1 の API で抽象化されている cde 型を

```
type 'a cde  = 'a code
```

と具体化して，. <> . や . ∼ といった BER MetaOCaml のステージ化注釈を部分的に直接扱う．. <> . は OCaml の型付きコードテンプレートを生成する演算子で，. ∼ は新しいコードテンプレートの一部に既存のコードテンプレートを展開する演算子であり，OCaml のコードテンプレートに付与される型が code 型である．

ここでは典型的なストリーム処理の例として，与えられたログファイルに grep のような検索をかけ，指定されたキーワードを含む行のみ行番号を添えて出力する処理を取り上げる．この処理をストリーム処理として表現するには，与えられたログファイルの各行を要素とするストリームを生産し，指定語句を含む要素で filter することを考えればよい．

まずは，ファイル名を受け取ってそのファイルの各行を要素とするストリームを生産する関数 of_file

```
val of_file : string → string stream
```

を用意する．この関数は strymonas の API では提供されていないが，次のように unfold を用いて容易に実装できる．なお，of_file は種々の副作用を扱っているがこれは対象コード上での話であり，of_file 自体は純粋な関数である．

```
let of_file filename =
  let step ch ondone = fun k → .<
    try
      let line = input_line .∼ch in
      .∼(k .<line>.)
    with End_of_file →
      (close_in .∼ch; .∼ondone)
  >. in
  let z = .<open_in filename>. in
  unfold step z
```

一般に，unfold とは状態遷移関数 step と初期状態 z の順序対 (step, z) で表現されるストリームを生産する関数のことである．このような順序対による表現は数学的なストリームの定義で用いられ，各々次のようなインターフェースを持つ．

```
val step : 'z → ('a * 'z) option
val    z : 'z
```

ここで，'a はストリームから取り出される要素の型に相当し，step は現在の状態 z_i を与えらえると，そのときの要素 x_i と次の状態 z_{i+1} を返すことでストリームを状態遷移させ，要素がなくなった場合は None を返して停止させる．

しかしながら，strymonas の unfold には，生産されるストリームの状態遷移関数 step を命令的かつ CPS で表現し，これと初期状態 z をステージ化したものが与えられる．これらは次のインターフェースをもつ．

```
val step : 'z cde → unit cde → ('a cde) emit
val    z : 'z cde
```

unfold の第 1 引数となるのが step であり，上記のコードにおける step は，ファイルとしてのストリームの状態を表す入力チャネルのコード ch と，停止時に起こなう終了処理の付随的な一部を表すコード ondone を受け取って，継続 k を引数にもつ CPS の関数 (図 1 の 3 行目記載の emit 型の値) を返す関数である．この継続 k はパイプラインで後続する変換者と消費者のセマンティックアクションを合成したセマンティックアクションをステージ化したものであるため，of_file の生産するストリームの step 関数が返す CPS の関数は，input_line によって ch が保持する入力チャネルからの 1 行分の読み込みと読み込み位置の更新に挑戦し，成功した場合には読み込まれた値 line を用いて引数の k が保持する処理を行い，失敗 (ファイル終端に到達) した場合にはチャネルを閉じたり ondone による付随的な終了処理を行うコードを生成するように定義されている．

また，unfold の第 2 引数となるのが z であり，上記のコードでは，of_file の引数で指定されたファイルの先頭を指している入力チャネルを表すコードとして定義されている．これが step 関数に ch として渡され利用される．

次に，第 1 引数中に第 2 引数の文字列が含まれているかどうか判定する関数をステージ化した関数 contain

```
val contain : string cde → string → bool cde
```

を用意する．部分文字列の判定処理には OCaml の標準的なライブラリで提供される Str モジュールの関数を使えるため，この関数は次のように書ける．

```
let contain s1 s2 = .<
  let re = Str.regexp_string s2 in
  try
     ignore (Str.search_forward
                          re .~s1 0);
     true
  with
  | Not_found → false
  >.
```

以上で定義した of_file 関数と contain 関数に加え，読み込み対象となるログファイルの名前 $filename$ と指定語句 $keyword$ を用いて，上述のストリーム処理は次のように書くことができる．

```
zip_with pair
  (iota (int 1))
  (of_file filename)
|> filter (fun e →
            contain .<snd .~e>. keyword)
|> take (int 10)
|> iter (fun e →
          .<Printf.printf "%d: %s\n"
            (fst .~e) (snd .~e)>.)
```

iota でファイルの行数を表現する 1 以上の数値のストリームを生成し，これを of_file で生成されたファイルのストリームと zip_with によって組み合わせることでタプルのストリームを生成し，$keyword$ の条件を満たしている要素のみ filter で抽出し，それらを iter に渡して標準出力に書き出すことで全体の処理は構成されている．ただし，あまりにも大量の行が出力されると不便なので，take によって出力行数が最大 10 件となるように制限した．このパイプラインによって生成されるコードを次に示す．

```
let lv_1 = 10 − 1 in
let lv_2 = ref 1 in
let lv_3 = open_in filename in
let lv_4 = ref true in
let lv_5 = ref 0 in
```

```
while ((! lv_5) ≤ lv_1) && (! lv_4) do
  let lv_6 = ! lv_2 in
  incr lv_2;
  (try
      let line_7 = input_line lv_3 in
      if
        let re_8 = Str.regexp_string
                                   keyword in
        try
          ignore (Str.search_forward
                re_8 (snd (lv_6, line_7)) 0);
          true
        with
        | Not_found → false
      then
        (incr lv_5;
         Printf.printf "%d: %s\n"
          (fst (lv_6, line_7))
          (snd (lv_6, line_7)))
      else ()
    with
    | End_of_file → (close_in lv_3;
                     lv_4 := false))
done
```

このコードの品質はそれほど悪くないものの，主に次の 2 点で大きな改善の余地をもつ．

1 つ目は，ループの度に *keyword* から毎回同じ正規表現 Str.regexp_string *keyword* を生成していることである．これは正規表現の生成と let 束縛を大域的な位置で行うように移動することで解決でき，そのためには図 2 の EDSL で提供されている glet(2 行目) が利用できる．glet を用いて，contain 関数は次のように書き換えられる．

```
let contain s1 s2 =
  let re = glet .<Str.regexp_string s2>. in
  .<
    try
      ignore (Str.search_forward
                               .~re .~s1 0);
      true
    with
    | Not_found → false
  >.
```

2 つ目は，zip_with のセマンティックアクションがタプルを固定サイズの中間データ構造として用いるように指定していることである．これを解決するには現状 strymonas が提供する低水準の API(第 4 節，図 7) を直接利用する必要があり，具体的には zip_raw, filter_raw，map_raw’，fold_raw という関数を利用することになる．

以上に基づいた新しいパイプラインは，殆ど上記のパイプラインと同様に，

```
zip_raw
  (iota (int 1))
  (of_file filename)
|> filter_raw (fun (_, e2) →
              contain e2 keyword)
|> zip_raw (from_to (int 1) (int 10))
|> map_raw' snd
|> fold_raw (fun (e1, e2) →
            .<Printf.printf "%d: %s\n"
              .~e1 .~e2>.)
```

と記述される．このパイプラインによって生成されるコードを次に示す．

```
let lv_8 = Str.regexp_string keyword in
let lv_1 = ref 1 in
let lv_2 = ref 1 in
let lv_3 = open_in filename in
let lv_4 = ref true in
while ((! lv_1) ≤ 10) && (! lv_4) do
  let lv_5 = ! lv_2 in
  incr lv_2;
  (try
      let line_6 = input_line lv_3 in
      if
        try
          ignore (Str.search_forward
                    lv_8 line_6 0);
          true
        with
        | Not_found → false
      then
        let lv_7 = ! lv_1 in
        (incr lv_1;
         Printf.printf "%d: %s\n" lv_5 line_6)
      else ()
    with
    | End_of_file → (close_in lv_3;
                     lv_4 := false))
done
```

このコードは上記の 2 点の改善だけではなく 10 − 1 が 9 に丸め込まれる最適化が行われていたりするなど，手書きで高効率になるように記述されたストリーム処理とそれほど遜色ない品質に改善されている．唯一無駄な lv_4 という変数が除去されていないが，これを解決するためには unfold を低水準の API で置き換える必要があり，本論文では割愛する．

```
let s_3    = ref 0    in  let arr_4  = arr1     in  let i_5 = ref 0 in
let curr_6 = ref None in  let nadv_7 = ref None in
let adv_16 () =
  curr_6 := None;
  while ((! curr_6) = None) && (((! nadv_7) ≠ None) || ((! i_5) ≤ ((Array.length arr_4) − 1))) do
    match ! nadv_7 with
    | Some adv_8 → adv_8 ()
    | None  →
        let el_9 = arr_4.(! i_5) in
        incr i_5;
        let arr_10 = arr2 in
        let i_11 = ref 0 in
        let old_adv_12 = ! nadv_7 in
        let adv1_15 () =
          if (! i_11) ≤ ((Array.length arr_10) − 1) then
            let el_13 = arr_10.(! i_11) in
            let t_14 = el_9 * el_13 in
            (incr i_11; curr_6 := (Some t_14))
          else nadv_7 := old_adv_12 in
        nadv_7 := (Some adv1_15); adv1_15 ()
  done in
adv_16 ();
let arr_17 = arr2 in
let i_18 = ref 0 in
let term1r_19 = ref ((! curr_6) ≠ None) in
let nr_20 = ref 20000000 in
while ((! nr_20) > 0) && ((! term1r_19) && ((! i_18) ≤ ((Array.length arr_17) − 1))) do
  let el_21 = arr_17.(! i_18) in
  incr i_18;
  let arr_22 = arr1 in
  let i_23 = ref 0 in
  while ((! nr_20) > 0) && ((! term1r_19) && ((! i_23) ≤ ((Array.length arr_22) − 1))) do
    let el_24 = arr_22.(! i_23) in
    let t_25 = el_21 − el_24 in
    incr i_23;
    match ! curr_6 with
    | Some el_26 →
        (adv_16 (); term1r_19 := ((! curr_6) ≠ None);
         decr nr_20; s_3 := ((! s_3) + (el_26 + t_25)))
  done
done;
! s_3
```

図 3　zip_flat_flat_v1($arr1$，$arr2$ が引数の関数)

3　実験

本研究では，次の実験環境を用いて strymonas v2 の性能を実際に評価した．

実験環境: 1.8 GHz デュアルコア Intel Core i5，8 GB 1600 MHz DDR3，macOS Catalina 10.15.4，BER MetaOCaml N107 (native backend)

実験では ocamlopt -O2 -unsafe -nodynlink でコンパイルされたコードを 30 回実行し，実行に要した時間の平均値を計測した．また，各コードの実行前には GC を強制的に実行した．各パイプラインの入力には以下の配列を利用したが，これらはコードの実行前に確保しておくことで計測の時間には含めなかった．

```
let v    = Array.init
             100_000_000 (fun i → i mod 10)
let vHi  = Array.init
              10_000_000 (fun i → i mod 10)
let vLo  = Array.init 10 (fun i → i mod 10)
let vFaZ = Array.init     10_000 (fun i → i)
let vZaF = Array.init 10_000_000 (fun i → i)
```

```
1  let lv_11 = (Array.length arr1) − 1 in  let lv_3 = (Array.length arr2) − 1 in
2  let lv_17 = (Array.length arr2) − 1 in  let lv_4 = (Array.length arr1) − 1 in
3  let lv_7  = ref 0 in  let lv_8 = ref 0 in  let lv_9 = ref false in
4  let lv_10 = ref (Obj.obj (Obj.new_block 0 0)) in  let lv_12 = ref (Obj.obj (Obj.new_block 0 0)) in
5  let lv_13 = ref 0 in  let i_14 = ref 0 in
6  while ((! i_14) ≤ lv_4) && ((! lv_13) ≤ 19999999) && ((! lv_9) || ((! lv_8) ≤ lv_3)) do
7    let iv_15 = ! i_14 in
8    incr i_14;
9    let el_16 = Array.get arr1 iv_15 in
10   let i_18 = ref 0 in
11   while ((! i_18) ≤ lv_17) && ((! lv_13) ≤ 19999999) && ((! lv_9) || ((! lv_8) ≤ lv_3)) do
12     let iv_19 = ! i_18 in
13     incr i_18;
14     let el_20 = Array.get arr2 iv_19 in
15     let lv_21 = el_16 * el_20 in
16     let lv_22 = ref true in
17     while (! lv_22) && ((! lv_9) || ((! lv_8) ≤ lv_3)) do
18       if not (! lv_9) then
19         (let el_25 = Array.get arr2 (! lv_8) in
20          incr lv_8; lv_10 := el_25; (); lv_12 := 0; lv_9 := true);
21       if ! lv_9 then
22         (if (! lv_12) ≤ lv_11 then
23            let el_23 = Array.get arr1 (! lv_12) in
24            let lv_24 = (! lv_10) − el_23 in
25            incr lv_12; lv_22 := false; incr lv_13; lv_7 := ((! lv_7) + (lv_21 + lv_24))
26          else lv_9 := false)
27     done
28   done
29 done;
30 ! lv_7
```

図 4　zip_flat_flat_v2(*arr*1，*arr*2 が引数の関数)

3.1　コード生成の改善

まずは，strymonas v1 において固定サイズの中間データ構造とクロージャを含むコードが生成されてしまうパイプラインの 1 つ zip_flat_flat(第 3.2 節参照)

```
zip_with (+)
  (of_arr (int_array arr1)
   |>  flat_map (fun x →
         of_arr (int_array arr2)
         |> map (fun y → x * y)))
  (of_arr (int_array arr2)
   |>  flat_map (fun x →
           of_arr (int_array arr1)
           |> map (fun y → x − y)))
|> take (int 20_000_000)
|> fold (+) (int 0)
```

を取り上げ，strymonas v2 によってこのパイプラインから生成されるコードが改善されていることを確認する．strymonas v1 と v2 の各々が zip_flat_flat から生成するコード zip_flat_flat_v1 及び zip_flat_flat_v2 を，図 3 と図 4 に示す (図 4 の 4 行目記載の Obj.obj (Obj.new_block 0 0) は第 4.2 節を参照)．図 4 のコードには，図 3 の 2 行目で定義されている cuur_6 や nadv_7 のような参照セル内で扱われる option 型の値 (固定サイズの中間データ構造) と，14 - 19 行目で定義されているクロージャadv1_15 が存在していないため，第 1 節の問題 2. が解決されていることが分かる．

次に，これらのコードに対する実験の結果を図 5 に示す †2．縦軸は計測された実行時間の平均値を表しており，バーが低いほど高性能である．実験において，パイプラインの入力値となる *arr*1 と *arr*2 にはそれぞれ v と vLo を用い，第 1 節の問題 2. の解決がどの程度パフォーマンスに影響するのか測定した．実行結果を比較すると，zip_flat_flat に対する strymonas v2 の生成コードは v1 のものより 5 倍程度高速化さ

†2　strymonas v1 は BER MetaOCaml N104 でしかサポートされていないが，N104 で生成したコードを N107 で実行した．

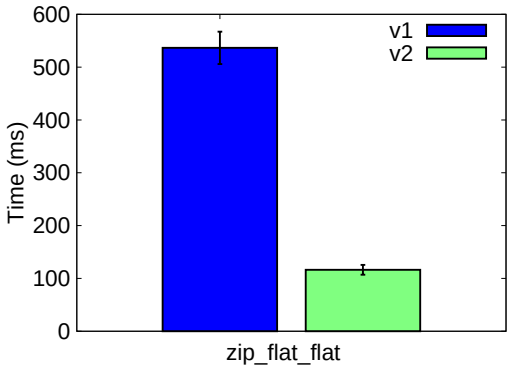

図 5　strymonas v1, v2 のパフォーマンス比較: エラーバーは 95%信頼区間

れており，問題の解決によって実際にパフォーマンスが向上するという結果を得られた．

3.2　ベンチマーク

本研究では，次のベンチマークを用いて strymonas v2 による生成コードの性能を測る実験を行った．

- **sum**: $\sum_i x_i$，配列の要素 (int 型) の和
- **sumOfSquares**: $\sum_i {x_i}^2$，配列の要素の二乗和
- **sumOfSquaresEven**: 配列の偶数要素のみの二乗和
- **maps**: 配列に自然数倍する複数の map をかけるパイプライン
- **filters**: 配列に自然数と比較する複数の filter をかけるパイプライン
- **cart**: $\sum_i \sum_j x_i y_j$，x_i を要素とする配列に flat_map を利用
- **dotProduct**: 2 つの配列のドット積，zip_with (×) を利用
- **flatMap_after_zipWith**: $\sum_i \sum_j (x_i + x_i) y_j$，zip_with (+) 後に flat_map
- **zipWith_after_flatMap**: 変換されない配列と flat_map された配列の zip_with (+)
- **flat_map_take**: flat_map 後に take するパイプライン
- **zip_filter_filter**: filter された 2 つの配列の zip_with (+)
- **zip_flat_flat**: flat_map された 2 つの配列の zip_with (+)，第 3.1 節のパイプライン

これらのベンチマークは [8] で用いられているものと，[11] の zip_filter_filter と zip_flat_flat に由来する．

本実験では上記のベンチマークに対し，strymonas v2 による生成コード (staged)，手書きで記述された効率的なコード (baseline)，streaming [5] という OCaml のストリームライブラリを用いたコード (source) を用意して，これら 3 つのコードの実行速度を測定した．streaming [5] は，pull と push という 2 つのモデルに基づいたストリーム処理をそれぞれ提供するが，ここでは pull 式のストリーム処理が実装されている Source モジュールを用いた [†3]．ただし flat_map が実装されていないため，Source モジュールでいくつかのベンチマークは実行できなかった．入力に関しては，各々のベンチマークについて次のように設定した．

- **sum**: v
- **sumOfSquares**: v
- **sumOfSquaresEven**: v
- **maps**: v
- **filters**: v
- **cart**: vHi と Vlo
- **dotProduct**: vHi と vHi
- **flatMap_after_zipWith**: vFaZ と vFaZ
- **zipWith_after_flatMap**: vZaf と vZaf
- **flat_map_take**: v と Vlo
- **zip_filter_filter**: v と VHi
- **zip_flat_flat**: v と Vlo

実験結果を図 6 に示す．縦軸は計測された実行時間の平均値を表しており，バーが低いほど高性能である．strymonas v2 と手書きのコードはおおよそ同程度のパフォーマンスとなり，これらは streaming よりも約 5 - 25 倍速く処理を終えるという結果が得られた．

†3　最新の streaming 全体は BER MetaOCaml N107 ではサポートされていないので，Source モジュールだけを取り出して利用した．

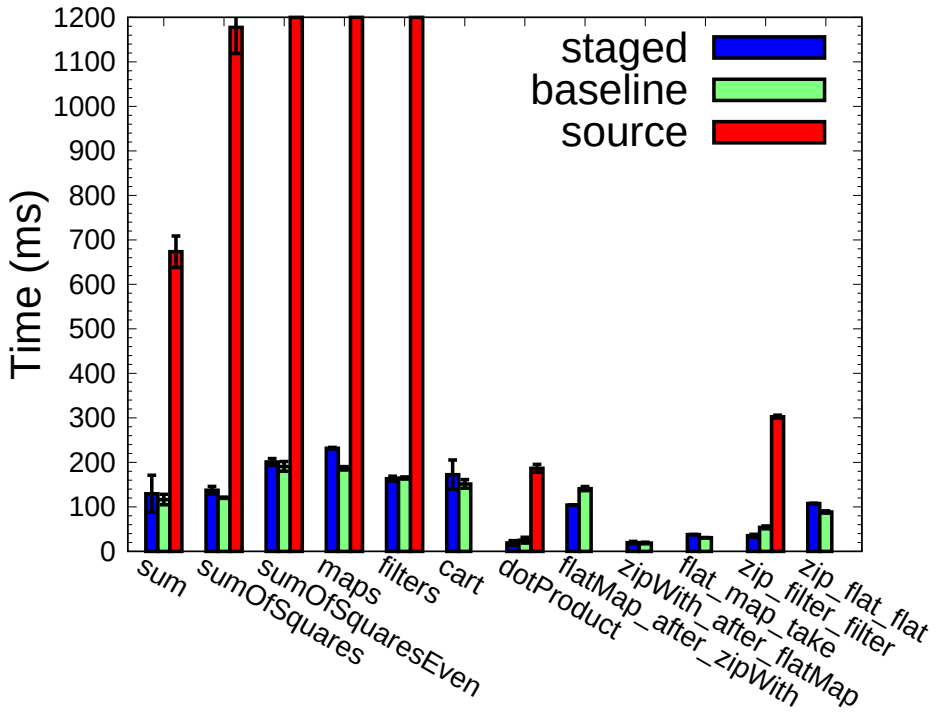

図 6　strymonas v2(staged)，手書き (baseline)，streaming(source) のパフォーマンス比較: エラーバーは 95%信頼区間，source の sumOfSquaresEven，maps，filters にかかった時間はそれぞれ約 1.5，5.0，3.8 秒

4　実装

第 2.1 節ではユーザ向けの高水準な API(図 1) を紹介したが，これらはより低水準な API によって実装されている．strymonas が提供する低水準の API を図 7 に示す．低水準 API はより一般的な表現の関数を提供するため，高水準 API よりも提供する演算子が少ないなどの特徴を持つ．例えば，take に対応した take_raw のような関数は低水準 API には存在せず，高水準 API の take は

```
take n s
= map_raw' snd (zip_raw (from_to (int 1) n) s)
```

という恒等式を利用して実装されている．このように実装された take は zip(_raw) を用いた定義の見かけに反し，タプルのような中間データ構造を生成コードに含まない (例: 第 2.2 節の最後の生成コード)．

4.1　中間言語

strymonas におけるストリームは'a stream 型の値であるが，その実体はコード生成に用いられる中間言語のコードである．この中間言語は GADT の st_stream という型で定義され，メタ言語に埋め込まれている．ここで，strymonas v2 の中間言語を図 8 に示す．12 行目の Init は変数の初期化・宣言文であり，13 行目の Lin が主に生産直後のストリーム，14 - 17 行目の Filtered/Stuttered，Break，Nested は各々順番に，continue 文を含んだループ，break 文を含んだループ，多重のネストしたループによって要素が取り出されるように変換がかけられたストリームを定義する構文である．Filtered/Stuttered，Break，Nested は低水準 API の filter_raw，zip_raw，flat_map_raw に各々対応する．

strymonas v1 では，Init，Filtered/Stuttered，Break に対応する明示的な構文が存在しない未洗練の中間言語を利用していたために，第 1 節で述べ

```
type 'a st_stream
type goon = bool cde

(* 生産者 *)
val pull_array       : int cde → (int cde → 'a emit)        → 'a st_stream
val initializing     : 'z cde → ('z cde → 'a st_stream)     → 'a st_stream
val initializing_ref : 'z cde → ('z ref cde → 'a st_stream) → 'a st_stream
val initializing_arr : 'z cde array → ('z array cde → 'a st_stream) →
                       'a st_stream
val infinite : 'a emit → 'a st_stream
val guard    : goon → 'a st_stream → 'a st_stream

(* 変換者 *)
val map_raw      : ('a → ('b → unit cde) → unit cde) →
                   'a st_stream → 'b st_stream
val map_raw'     : ('a → 'b)       → 'a st_stream → 'b st_stream
val filter_raw   : ('a → bool cde) → 'a st_stream → 'a st_stream
val flat_map_raw : ('a cde → 'b st_stream) → 'a cde st_stream →
                   'b st_stream
val zip_raw      : 'a st_stream     → 'b st_stream → ('a * 'b) st_stream

(* 消費者 *)
val fold_raw : ('a → unit cde) → 'a st_stream → unit cde
```

図 7　strymonas の低水準 API

```
type 'a pull_array = {upb: int cde;
                      index: int cde → 'a emit}

type 'a init =
  | Ilet : 'a cde → 'a cde init
  | Iref : 'a cde → 'a ref cde init

type 'a producer =
  | For       : 'a pull_array → 'a producer
  | Unfold    : 'a emit       → 'a producer
 and _ st_stream =
  | Init      : 's init * ('s → 'a st_stream)              → 'a st_stream
  | Lin       : 'a producer                                → 'a st_stream
  | Filtered  : ('a → bool cde) * 'a producer              → 'a st_stream
  | Stuttered : 'a option producer                         → 'a st_stream
  | Nested    : 'b cde st_stream * ('b cde → 'a st_stream) → 'a st_stream
  | Break     : goon * 'a st_stream                        → 'a st_stream
 and 'a stream = 'a cde st_stream
```

図 8　strymonas の中間言語

た問題 1. が発生していた．strymonas v2 では，ループの構造へ影響を持たない入力パラメータの変数や continue 文，break 文，ネストしたループという全く異なる構造のループで要素が取り出されるストリームを図 8 のように各々独立した構文で表現できるようにすることで，表現力の向上やコード生成で代数的に扱いやすくなる（パターンマッチングや再帰関数で処理しやすい）といった中間言語の洗練化がなされた．

ただし，Filtered/Stuttered や Break はそれぞれ continue 文や break 文に対応して値が取り出されるように変換されたストリームを表現するため，これらから OCaml のように continue 文や break 文を持たない言語のコードを生成する場合には少し工夫が必要である．次に命令的な擬似コードを使ってその例

を示す．例えば Filtered/Stuttered で表されるストリームを作ってから消費するパイプライン

$stream$ |> filter g |> fold f z

の場合には，

```
accum = z;
for e_i in stream {
  if g(e_i) {
    accum = f(accum, e_i);
  }
}
return accum;
```

のように，g を満たさないかを確認して continue で処理を飛ばす場合とは逆に，g を満たすかを確認して処理を続けるようにすることで，continue 文付きのループを模倣できる．また，Break で表されるストリームを作ってから消費するパイプライン

$stream$ |> take n |> fold f z

の場合では，

```
accum = z;
goon = true;
for (e_i in stream) && goon {
  if i ≥ n {
    goon = false;
  } else {
    accum = f(accum, e_i);
  }
}
return accum;
```

のように，goon のような補助変数をループ継続判定のガードに用いることで，break 文付きのループを模倣できる．

4.2 固定サイズの中間データ構造の除去

第 1 節の問題 2. の原因となるのは，flat_map されたストリーム (Nested) の対を zip するパイプラインのコード生成を行う場合に，生成するコードで未初期化のミュータブル変数を利用しなければならないことである．関数型言語ではこれを ref None のように option 型の値への参照として表現する方法が自然であり，strymonas v1 はこの方式を採用していたため option 型の値という固定サイズの中間データ構造が生成コードに含まれてしまっていた (図 3 の curr や nadv)．

このような固定サイズの中間データ構造の使用を避けるためには，対象言語のより低水準な機能をつかって未初期化のミュータブル変数を実現する必要がある．OCaml を対象言語とする場合は図 4 で見られたが，

```
ref (Obj.obj (Obj.new_block 0 0))
```

というように，Obj モジュールで生成した値を未初期化時に参照する表現で，option 型の値を用いず未初期化のミュータブル変数を表すことができる．

4.3 クロージャの除去

strymonas では，Filtered/Stuttered や Nested のストリーム同士を zip する際に，前処理として片方のストリームに線形化と呼ばれる，ストリームの表現が Lin となるよう変換する正規化の処理を行う必要がある．Nested は flat_map(_raw) によって生成されるストリームであり，コード生成にクロージャが関係してくるのはストリームが Nested の場合である．したがって，本節では Nested の線形化について説明する．

今，サイズ N で i 番目の要素が e_i のストリームを $[e_1, e_2, \ldots, e_N]$ のように表記すると，例えば

```
[x_1, x_2, ..., x_N]
|> flat_map (fun x →
      [x+y_1, x+y_2, ..., x+y_N']
```

という演算から生成されるストリームは，

```
[x_1+y_1, x_1+y_2, ..., x_1+y_N',
 x_2+y_1, x_2+y_2, ..., x_2+y_N',
 ...,
 x_N+y_1, x_N+y_2, ..., x_N+y_N']
```

となる．ここで $[x_1, x_2, \ldots, x_N]$ のように，flat_map の対象となるストリームを外側のストリームと呼ぶことにし，外側のストリームの要素が x のときに flat_map のセマンティックアクションによって生成することができる $[x+y_1, x+y_2, \ldots, x+y_{N'}]$ のようなストリームを内側のストリームと呼ぶことにする．Nested を線形化する場合，外側のストリームの現在の要素 x などを自由変数として含む内側のストリームの状態遷移を担う関数を生成するコードに用いざるを得ない．strymonas v1 ではこのような状態遷移の関数を生成する対象コードにクロージャとしてそのまま埋め込んでいたため (例: 図 3 の adv1_15)，第 1 節の問題 2.2. が起きていた．これを避けるためには，

strymonas で状態遷移の関数にクロージャ変換を行う必要があり，以下ではその概要を説明する．

クロージャ変換の基本は，自由変数を含む関数に対してその自由変数を引数として束縛させる変換をかけ，その関数が定義されたときの自由変数の値を呼び出し時に参照できるよう保存しておくことである．strymonas v2 のクロージャ変換は，大域的な位置に let 束縛された参照セル内に自由変数となる外側のストリームの現在の要素 x などを保存するようにして (例: 図 4 の lv_10 と lv_12)，内側のストリームの状態遷移の関数が任意の位置から x などを参照できるようにすることである †4．この参照セルは x などの値を保存する以前から割り当てられるので未初期化のミュータブル変数として表現する必要があり，そこで第 4.2 節の

```
ref (Obj.obj (Obj.new_block 0 0))
```

が利用される．

以上のような処理に加え種々の関数をインライン展開するなどして，全体のクロージャ除去の処理が果たされる．ここで行われるクロージャ除去の処理は安全であり，すなわち strymonas が保証する安全なコードの生成が厳格に守られる．

5 関連研究

streaming [5] はコンスタントメモリ，高パフォーマンス，安全なリソース管理などが特徴のストリーム処理を提供する OCaml のライブラリである．これらは stream fusion の利用によって実現されているものの，streaming では strymonas のようなステージ化を用いた高度な最適化などは行われないため，タプルや option 型の値などの固定サイズの中間データ構造やクロージャなどに起因した種々のオーバーヘッドが存在する．

stream fusion が施されたストリームライブラリのうち，オブジェクト指向言語で関数型のストリーム処理を実現する実用的なライブラリとして有名であるのが Java 8 streams である．オブジェクト指向言語における stream fusion の研究において，Java 8 streams はその題材として盛んに取り上げられており，そのような研究には [2] などがある．

Weld IR [10] は TensorFlow，Apache Spark，NumPy，Pandas といったデータ分析ライブラリを跨いでクロスプラットフォームなバッチ処理の最適化を実現する中間表現であり，Kroll ら [9] は Weld IR にストリーム処理の表現能力を加えて拡張した Arc という中間表現を提案している．

6 まとめと今後の課題

本研究では，当初の strymonas が抱えていた中間言語や例外的に残留する固定サイズの中間データ構造とクロージャに関する問題を解決するため，中間言語の再設計や生成コードに対するより高度な最適化の実装を行った．この改善が施された strymonas は実験的にその性能の高さが証明されており，zip と flat_map を含むような複雑なものを含め API で表現可能な任意のパイプラインに対し，中間データ構造やクロージャを用いないストリーム処理を行う安全なコードを生成できるようになった．これによって，高水準かつ高性能なストリームライブラリの基盤となるストリーム処理のコード生成器が実現された．

今後の課題としては 3 つが考えられる．1 つ目は，strymonas で window 処理をサポートすることである．window 処理とは，入力ストリームを複数の有限データ列 (ウインドウ) に分割し，並行なバッチ処理によってそれらを処理する機能である．window 処理は実用的な高機能ストリームライブラリの中核となる機能であり，特定の対象言語に依存しない高性能で安全な window 処理のコードの生成器を実現することは大きな課題である．2 つ目は，strymonas の中間言語を C/LLVM へコンパイルすることである．コード生成の対象が OCaml のような高水準言語となる場合，言語機能の問題で達成できるストリーム処理の性能に限界がある．例えば OCaml は break 文をサポートしないため，take を行うストリーム処理のコードでは第 4.1 節の最後で紹介したように補助変数のガードを利用して break の挙動を模倣すること

†4 ある内側のストリームの状態遷移の関数のクロージャは同時には 1 つまでしか存在しないため，各自由変数に対して 1 つずつ参照セルを導入すればよい．

になる．そこで，C や LLVM のような低水準の計算が行える言語であればこのような制約を受けず，より効率の良いストリーム処理を実現できるはずである．幸い strymonas v2 ではコード生成の対象言語を cde 型によって抽象化するようになったため，図 2 のインターフェースを満たす C/LLVM 向けの EDSL を実装すればこれを実現できる．3 つ目は，TensorFlow などのデータ分析フレームワークを strymonas の立場から考えることである．生成するストリーム処理のコードの安全性やコンパイラによらない効率性を保証する strymonas の最適化は有用であり，このアプローチがバッチ処理の方面に応用されることは大変望ましい．

参 考 文 献

[1] Biboudis, A.: *Expressive and Efficient Streaming Libraries*, PhD Thesis, University of Athens, Athens, March 2017.

[2] Biboudis, A., Palladinos, N., Fourtounis, G., and Smaragdakis, Y.: Streams a la carte: Extensible Pipelines with Object Algebras, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, Vol. 37, 2015, pp. 591–613.

[3] Coutts, D., Leshchinskiy, R., and Stewart, D.: Stream Fusion: From Lists to Streams to Nothing at All, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, New York, NY, USA, ACM, 2007, pp. 315–326.

[4] Coutts, D. and Stewart, D.: stream-fusion: Faster Haskell lists using stream fusion, `https://hackage.haskell.org/package/stream-fusion`.

[5] I, R.: odis-labs/streaming: Fast, safe and composable streaming abstractions., `https://github.com/odis-labs/streaming`.

[6] Kiselyov, O.: The Design and Implementation of BER MetaOCaml, *Functional and Logic Programming*, Springer, 2014, pp. 86–102.

[7] Kiselyov, O., Biboudis, A., Palladinos, N., and Smaragdakis, Y.: Strymonas Streams: stream fusion, to completeness, `https://strymonas.github.io/`.

[8] Kiselyov, O., Biboudis, A., Palladinos, N., and Smaragdakis, Y.: Stream fusion, to completeness, *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, New York, ACM Press, January 2017, pp. 285–299.

[9] Kroll, L., Segeljakt, K., Carbone, P., Schulte, C., and Haridi, S.: Arc: An IR for Batch and Stream Programming, *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages*, DBPL 2019, New York, NY, USA, Association for Computing Machinery, 2019, pp. 53–58.

[10] Palkar, S., Thomas, J. J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S., and Zaharia, M.: Weld: A Common Runtime for High Performance Data Analytics, *The biennial Conference on Innovative Data Systems Research*, CIDR '17, Jan 2017.

[11] Parreaux, L., Shaikhha, A., and Koch, C.: Quoted Staged Rewriting: A Practical Approach to Library-Defined Optimizations, *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE ' 17)*, (2017), pp. 15.

[12] Rompf, T. and Odersky, M.: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, *Commun. ACM*, Vol. 55, No. 6(2012), pp. 121–130.