法政大学学術機関リポジトリ
HOSEI UNIVERSITY REPOSITORY

PDF issue: 2025-07-01

Implementing Data-Flow Fusion DSL on Clojure

桜井, 勇貴 / SAKURAI, Yuuki

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
11
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2016-03-24
(URL)
https://doi.org/10.15002/00012879

Implementing Data-Flow Fusion DSL on Clojure

Yuuki Sakurai[†]
Graduate School of Computer and Information Sciences, Hosei University,
Tokyo, Japan
yuuki.sakurai@stu.hosei.ac.jp

***Abstract*—This paper presents a new optimization technique for programming language Clojure based on the standard fusion technique that merges list operations into a simplified loop.**

Short-cut fusions, foldr/build fusion, and stream fusions are standard fusion techniques used in a functional programming. Moreover, a recent fusion technique proposed by Lippmeier. Data flow fusion [7] can fuse loops containing several data consumers over stream operators on Haskell. However, this is only for Haskell. Clojure's transducers [2] are factory functions generating abstract list objects. The functions generated by them wait evaluation as a lazy evaluation partially.

We introduce data-flow fusion macros into Clojure as a dynamic typing mechanism and show the difference of data flow fusion between that of Clojure and Haskell.

We focus on Clojure which is a functional programming language with attracting features of dynamic typing, Lisp macros, partial lazy evaluation, and running on the Java Virtual Machine (JVM).

Our macro compiles S-expression of Clojure to Clojure code and Java class file. Our ideas are implemented as a domain specific language, which has strong points of the provision of a simple interface for loop fusion, and independence from the implementation of a Clojure Compiler.

We discuss the advantages and disadvantages of data-flow fusion macro from experimental results and adaptability of our macro.

I. Introduction

Loop fusion is a technique composing several loops to get rid of un-used data and duplicated loops containing data we can do without them. This method is widely practiced both automatically and manually.

A typical example of program fusion in a functional program is the map fusion or map distribution law as in equation (1).

$$\texttt{map}\ f \circ \texttt{map}\ g = \texttt{map}\ (f \circ g) \tag{1}$$

Though both sides give the same results, the RHS runs faster than the LHS. One reason for this is that the LHS calls the map function twice, which means the LHS executes loops twice, while the RHS does it once.

The second reason is LHS generates an intermediate list for carrying between `map` f and `map` g, while the RHS does not.

Basically, a program with no intermediate data structures nor simplified loops runs faster than the others. The purpose of the fusion technique is to transform a program that has no such features into a program that does.

Hand fusion is powerful and sometime the fastest way, but this causes the problems of lowering the readability of code. The modified code is often ugly and error prone.

Automated fusion prevents such kind of code degradation. However, automated fusion can be applied only to limited cases.

Our approach is the middle way between automatic and manual. The user of our system first insert data-flow fusion macro manually into the code, and then the Clojure macro processor takes care of fusion based on the data-flow information.

A. Clojure and functional programming languages

Clojure is a functional programming language optimized for parallel programming and executed on the JVM.

Functional programming languages, including Clojure, provide many useful functions for operation of lists generally. Typical list operations are map, filter, and reduce[1]. These operations return a newly constructed list without destroying the given list. Therefore, they consume both space and time. List operations in a functional program correspond to loops in an imperative program. So, the list operations appear frequently in a functional program. Their inefficiency can be a serious bottle neck in the performance of a functional program.

Lists in Clojure are a "sequence." The sequence is a super class for sequable (sequence-able) data types such as list, vector, and so on. The sequence is with the lazy evaluation. The list operation generates immutable lazy sequences.

B. Program Fusion on Clojure

The map distribution law obviously holds in Clojure, but loop fusions in general are complicated and not straight forward.

The case below is an example of filter-max implementation.

```
(defn filter-max [xs f pred]
  (let [ys (map f xs)
        zs (filter pred ys)
        max-value (reduce max 0 zs)]
    [zs max-value]))
```

And the next one is a fused code with loop-recur form.

```
(defn filter-max [xs f pred]
  (loop [[y & ys] xs max-value 0 zs []]
    (cond
     (not y)
     [(reverse ys) max-value]
     (pred y)
```

[†] Supervisor: Prof. Shuichi Yukita

[1] "map" function is mapping list to another list whose elements are applied by given function. "filter" function is filtering list that contains elements predicate returns true. "reduce" function apply binary operation between the all elements of the given list.

```
    (recur ys (max max-value y) (cons y zs))
    :else
    (recur ys y zs))))
```

Let n be the size of the list. The computational complexity of previous version is $O(3n)$, and the fused version is $O(2n)$. A Fused code is longer than the unfused version. A fused code loses some structural information in the unfused version.

A hand-fused program with loop-recur special form runs slower than the unfused program on Clojure, because map and reduce are defined as primitives and implemented as byte code. On the other hand, a hand-fused program is an S-expression with the loop-recur special form being compiled as byte code.

II. Related works

Transducers are given official functions for loop fusion on Clojure. The functions return the fused functions from the list operations and the compositions of them. The usage of these functions and a strategy of a fusion are to select and implement code for the fusion manually. The result of fusion is automatically generated.

Several techniques exist for the same purpose in programming languages Haskell and Gofer. Gill's foldr/build fusion [3] fuses two functions, the `folder` and the generator, which is called `build`, of the list automatically. Meijer's Hylomorphism [4] formalizes the fusion laws for algebraic(recursive) data types with Category theory and this automated fusion system was developed as a calculational fusion system HYLO [5].

However, they are not applicable for multiple consumer functions that contain several map/filter/reduce functions taking the same arguments.

Data flow fusion [7] compiles a program to a data flow language and fuses several loops of the same size. This technique can fuse the loops containing multiple consumer functions, but it focuses on the stream interface [6] in Haskell.

III. Data-flow fusion macro on Clojure

A domain specific language (DSL) is a method for abstracting program and a way to define another language for the specialists who is not a programmer. We implement a macro as a DSL for the users who don't know the program fusions.

Recursive functions are sometime complicated like as `goto` statement. They make a program bigger than a case without those functions [10]. The hand fusion of the list operations is time consuming especially when recursive calls are involved. Our macro enables to fuse several loops of list operations in the cases below.

- Simple fold-map, fold-filter fusion.
- Map-map, filter-map or map-filter fusion.
- A composition of above fusions.
- Multiple functions take the same argument from the single list.

A. Syntax and Semantics

The syntax is composed of let-form, map, filter, and reduce, because our macro aims at fusing multiple data flows while keeping the semantics the same as in Clojure.

The syntax of the macro is defined as below. The let expression plays the central role. The top level is $\langle \mathsf{exp} \rangle$.

```
(require [flow-fusion.flow-fusion])

(defn filter-max [vec1 f pred]
  (let [result
        (flow-fusion/flow-fusion
         (let [vec2 (map f vec1)
               vec3 (filter pred vec2)
               n (reduce max 0 vec3)]
           (list vec3 n)))]
    result))
```

Figure 1. Code for filter max

$\langle \mathsf{list\ operation} \rangle$ corresponds to Clojure's functions having the same names. List Operations are given as a syntax, not a function, but from programmers' view all of these look like ordinary Clojure functions. This is one of the strengths of our approach. `var` is a symbol. Let syntax is the same as Clojure's let. As the result, that semantics is a subset of Clojure, and there is no problems to read this macro.

$$
\begin{array}{rcl}
\langle \mathsf{DSL} \rangle & \longrightarrow & (\texttt{flow-fusion/flow-fusion} \\
 & & \quad \langle \mathsf{body} \rangle) \\
\langle \mathsf{body} \rangle & \longrightarrow & (\texttt{let}\ [\langle \mathsf{inner} \rangle]\ \langle \mathsf{body} \rangle) \\
 & | & (\texttt{list}\ \texttt{var} \cdots) \\
\langle \mathsf{inner} \rangle & \longrightarrow & \texttt{var}\ \langle \mathsf{apply} \rangle\ \langle \mathsf{inner} \rangle |\ \epsilon \\
\langle \mathsf{apply} \rangle & \longrightarrow & (\texttt{map}\ \texttt{exp}_{\texttt{clj}}\ \texttt{args} \cdots) \\
 & | & (\texttt{filter}\ \texttt{exp}_{\texttt{clj}}\ \texttt{arg}) \\
 & | & (\texttt{reduce}\ \texttt{exp}_{\texttt{clj}}\ \texttt{exp}_{\texttt{init}}\ \texttt{arg})
\end{array}
$$

Figure 1 is an example for our DSL code. The expression `flow-fusion/flow-fusion` is the special form for data flow fusion and its argument is the target code. The three loops, in the example, map, filter, and reduce are fused into a single loop by our macro.

B. Implementation

We implement the syntax as a macro via the traditional Lisp macro mechanism of Clojure. A Lisp macro is a simple function from S-expression to S-expression. An S-expression is basically composed of symbols and lists as in other languages of Lisp family. Clojure, in addition, has vectors as an element of the S-expression. Symbol contains all literals in that Lisp. However, the function of our macro contains side effects, because our transformation is from S-expression to Java byte code and S-expression.

There are two merits for the implementation as a macro of data-flow fusion. At first, the macro provides simple interface for programming as yet another syntax, or special form. The special syntax in Lisp is generalized as special forms, therefore, it is easy to understand and use for Lisp users. And second, the transformation independents on the implementation of Compiler and Interpreter, and it has been expressed as an external library.

IV. Algorithms for Transformation

The flow of our proposal technique is

1) A-Normalization, filter transformation, add create function, and type inference
2) Compile A-Normalized code to data flow process description

3) Schedule the data flow process description to abstract loop nests from
4) Concretize the abstract loop nests form to Java and Clojure Code
5) Compile java file and returns S-expression as Clojure macro.

The parts of our proposal technique for Clojure are the 1st step and the 4th step. In 1st step, we introduce the type inference to add create function, which is the label for the finally generated list. We redefine the last two steps for Clojure's S-expression and the JVM based code.

The part of step 2 to 3 derives from [7], but those steps are also redefined for Clojure. The purpose of these steps is to deform from the expression of composed functional code to procedural code in juxtaposition.

Data flow process description (DFPD) and abstract loop nests form (ALNF) are intermediate language for data flow fusion. The results of the fusion are a compiled Java class file and S-expression which is interface to call the compiled Java class file.

A. A-Normalization and filter transformation

A-Normalization [9] is a popular transformation technique from an expression-based functional program to an imperative program, such as assembler. This method transform to the expression that composite several inner expression by naming for each intermediate expression.

As the result, it regards the sequence of named expression by let syntax as a series of assign statement. Therefore, we can deformation from the let assign to assign statement.

In addition, we introduce the filter transformation. Filter transformation is

```
(let [ls1 (filter f1 ls0)] exp2)
⟹ (let [flag (map f1 ls0)]
     (mksel flag
       (fn [sel]
         (let [ls1 (pack sel ls0)] exp2)))).
```

The transformer applies this for each filter function recursively.

`mksel` is a guard form to execute the inner function when the nth element of `flag` is true. `pack` function is another filter function using every nth element of `ls0` as predicate.

B. add create function and type inference

Data flow fusion doesn't create intermediate data structures, but it creates returning data structures. The target DSL is a dynamic typed, but type information is required to determinate the type of the returning results.

Our technique use a simple type inference, which infers whether each variable is sequential (list) or not in the DSL code. The inference checks arguments and assigned variables of map, filter, and reduce special forms.

Our macro redefines a sequential variable in the returning results as

```
(list ls0 arg2 ...)
⟹ (let [new_ls0 (create ls0)]
     (list new_ls0 arg2...)))
```

with a let form and a create function.

```
(let [vec23453 (map inc vec1)]
  (let [flags3454 (map even? vec23453)]
    (mksel flags3454 (fn [sel3455]
      (let [vec33456 (pack sel3455 vec23453)]
        (let [n3457 (reduce max 0 vec33456)]
          (let [new-lseq3458 (create vec33456)]
            (list new-lseq3458 n3457))))))))
```

Figure 2. A Normalized Code for filter max

Figure 2 is the result of 3 steps, A-Normalization, filter transformation, and adding create function for filter max of Figure 1.

C. Data flow graph

Next, Our macro compiles from A-Normal form to data flow process description (DFPD), a data flow language. A code in that language generates a data flow graph. The syntax of DFPD is

$$
\begin{array}{rcl}
\langle \mathsf{dfpd} \rangle & \longrightarrow & (\langle \mathsf{operator} \rangle^*, \langle \mathsf{yeild} \rangle) \\
\langle \mathsf{operator} \rangle & \longrightarrow & (\texttt{mksel}\ [\texttt{flags sel}]\ \langle \mathsf{operator} \rangle^*) \\
& | & (\texttt{v}\ \leftarrow\ (\texttt{map exp ls})) \\
& | & (\texttt{v}\ \leftarrow\ (\texttt{pack sel ls})) \\
& | & (\texttt{n}\ \Leftarrow\ (\texttt{reduce exp}\ \texttt{v}_{\texttt{init}}\ \texttt{ls})) \\
& | & (\texttt{v}\ \leftarrow\ (\texttt{create ls})) \\
\langle \mathsf{yeild} \rangle & \longrightarrow & (\texttt{list var} \ldots).
\end{array}
$$

`flags`, `sel`, `ls`, `v`, `n`, `var` are variables and `exp`, $\texttt{v}_{\texttt{init}}$, are expressions of Clojure. The other symbols are special forms, the same meaning of flow-fusion DSL in the section III-A.

An edge of the graph is variable, and a node of the graph is special from. An example of the graph of filter max is in Figure 3.

D. Rate inference

The rate variable represents the abstract loop size and be a newly generated loop counter in an imperative program. Therefore, data flow fusion requires the rate (size) information as rate variables.

The algorithm of rate inference consists of the 4 steps as described below.

1) assign a symbolic size for each variables
2) make a restriction E (= $\{x_i = x_j, \ldots\}$).
3) resolve the restriction E.
4) assign rate variables for each a set of the same size.

The restriction of rate inference is

$$E = E_{map} \cup E_{pack} \cup E_{create}. \tag{2}$$

E_{map}, E_{pack}, E_{create} is

$$
\begin{array}{rcl}
E_{map} & = & \{\texttt{v} = \texttt{ls} \mid (\texttt{v}\ \leftarrow\ (\texttt{map exp ls})) \in D\} \\
E_{pack} & = & \{\texttt{v} = \texttt{sel} \mid (\texttt{v}\ \leftarrow\ (\texttt{pack sel ls})) \in D\} \\
E_{create} & = & \{\texttt{v} = \texttt{ls} \mid (\texttt{v}\ \leftarrow\ (\texttt{create ls})) \in D\}.
\end{array}
$$

The D is a set of expressions in DFPD and $\texttt{v} = \texttt{ls}$ is a restriction.

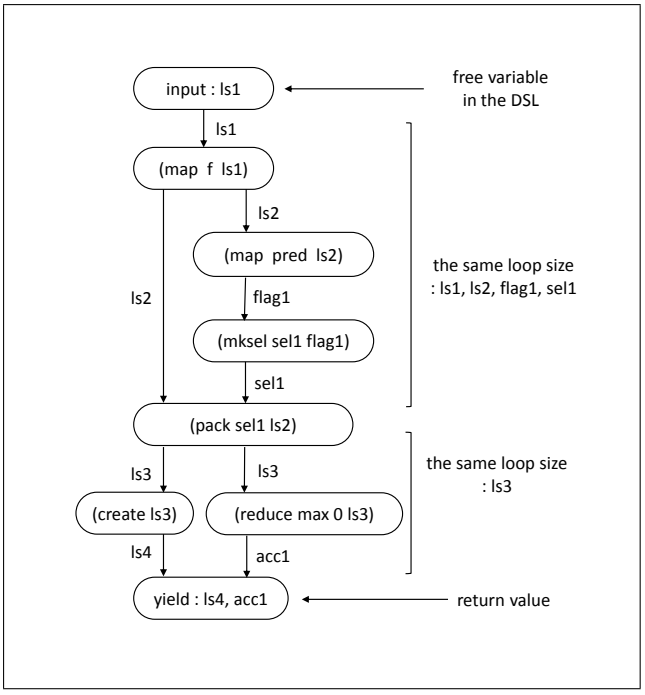

Figure 3. Data flow graph for filter max

E. Abstract loop nests form and process scheduling

The process scheduling fuses loops. Abstract loop nests form (ALNF) is loop-fused intermediate language and defined as

$$
\begin{array}{rcl}
\langle \mathsf{procedure} \rangle & \longrightarrow & (k,\ \langle \mathsf{nest} \rangle^*,\ \langle \mathsf{yeild} \rangle) \\
\langle \mathsf{nest} \rangle & \longrightarrow & (k,\ \langle \mathsf{start} \rangle^*,\ \langle \mathsf{body} \rangle^*,\ \langle \mathsf{inner} \rangle,\ \langle \mathsf{end} \rangle^*) \\
& | & (k,\ \langle \mathsf{body} \rangle,\ \langle \mathsf{end} \rangle) \\
\langle \mathsf{start} \rangle & \longrightarrow & \mathtt{x} = \mathtt{newVec}\ k \\
& | & \mathtt{n} = \mathtt{newAcc}\ \langle \mathsf{exp}_{\mathsf{init}} \rangle \\
\langle \mathsf{body} \rangle & \longrightarrow & \mathtt{x}_{\mathtt{elem}} = \mathtt{x}'_{\mathtt{elem}} \\
& | & \mathtt{x}_{\mathtt{elem}} = \mathtt{next}\ k\ \mathtt{ls}_{\mathtt{in}} \\
& | & \mathtt{x}_{\mathtt{elem}} = \langle \mathsf{exp}_{\mathsf{f}} \rangle \mathtt{x}_{\mathtt{elem}} \ldots \\
& | & \mathtt{acc} := \langle \mathsf{exp}_{\mathsf{f}} \rangle\ \mathtt{acc}\ \mathtt{x}_{\mathtt{elem}} \\
& | & \mathtt{writeVec}\ k\ \mathtt{vec}\ \langle \mathsf{exp}_{\mathsf{init}} \rangle \\
\langle \mathsf{end} \rangle & \longrightarrow & \mathtt{x} = \mathtt{read}\ \mathtt{acc} \\
& | & \mathtt{sliceVec}\ \langle \mathsf{exp}_{\mathsf{init}} \rangle \\
\langle \mathsf{yeild} \rangle & \longrightarrow & \langle \mathsf{exp}_{\mathsf{yeild}} \rangle
\end{array}
$$

That form is a composition of a loop and guards. A loop and guards are the abstractions of a `for` statement and `if` statements.

The Scheduling process is the third step and the main part for the fusion. The transformer of this part compiles DFPD to ALNF and fuse statements for each rate variable. DFPD is a sequence of statements, and the scheduler is restructuring statements. We define this transformer in Figure IV-E.

The function $\mathcal{T}$ is scheduler. The scheduler accepts a sequence of statements as DFPD and calls supplemental function $\mathcal{S}$ until the empty, nil. The transform function is $\mathcal{S}$, that decomposes a statement of DFPD and reconstructs from that to the nest of ALNF by $\triangleright$. f returns the corresponded rate variable of the variable. This data is in the section IV-D. $\triangleright$ is an operation to fuse for each statement with rate variable respectively.

F. Concretize code of Java and S-expression

In the last step, our macro generates Clojure and Java codes. After that, our macro compiles the concrete Java code to a JVM byte code, which is the finally fused code. The interoperation between Clojure and Java is to set the parameters in public static field variables. The concretization in Java is defined in Table I

A traditional Lisp macro is defined as a simple function from S-expression to S-expression. S-expression is a composition of two kinds of objects, symbols and lists. The result our data-flow fusion macro is also a function from S-expression to S-expression, but our macro invoke side effects, because of creating and compiling Java class file.

Figure 6 is the output of flow fusion in Java. `APersistentVector` is a class for vector of Clojure and `IFn` is a class for function in Clojure. The type of accumulators or the result of calculation is `Object` class, because Clojure is dynamic Typed. However, flag variables and rate variables are determinable. A flag variable requires to be boolean type and a rate variable requires to be integer type for loop counter. An object of `IFn` class uses with invoke method and it corresponds to the function calls of Clojure. The loop method in the compiled class is the fused loop.

Our macro calls `javac` command, the standard Java compiler with the generated Java code. It compiles a generated Java code such as Figure 6 and generates the class file. That class is the final result of the fusion.

Figure IV-F is an example from the filter-max code on Clojure. The part of `(.FlowFusionGenerated2675 vec1)` is to call for class member and `set!` sets the value at the variable. The flow in Figure IV-F is to set the values, loop parameters, to execute the loop and to return values.

The macro processor of Clojure with our macro expands the generated S-expression such as Figure IV-F in the defined (called) point as an S-expression.

```
(do
 (set! (.FlowFusionGenerated2675 vec1) vec1)
 (set! (.FlowFusionGenerated2675 fn2676) inc)
 (set! (.FlowFusionGenerated2675 fn2677) even?)
 (set! (.FlowFusionGenerated2675 fn2678) max)
 (.FlowFusionGenerated2675 loop)
 (list (.FlowFusionGenerated2675 new_lseq2660)
       (.FlowFusionGenerated2675 n2659)))
```

Figure 5. Generated Clojure code for filter max

V. Application

Filter max is an example for a composition of filter and reduce, and it exists universally in a functional program. In the situation using a sequence of data structure, such as XML, CSV, instances on the JVM, or some, filter corresponds to an extraction of data from the sequence and reduce corresponds to an integration of data. Additionally map is the control of modification of that data.

Neither build/fold fusion nor system HYLO can fuse the code of multiple reduces for the single list, but the data

$$\mathcal{T}(nest,\ \texttt{nil}) = nest$$
$$\mathcal{T}(nest,\ (\texttt{cons statement}_{\texttt{dfpd}}\ \texttt{statements}_{\texttt{dfpd}})) = \mathcal{T}(\mathcal{S}(nest)[\![\texttt{statement}_{\texttt{dfpd}}]\!], \texttt{statements}_{\texttt{dfpd}})$$

$$\mathcal{S}(nest)[\![(\texttt{mksel}\ [\texttt{flag sel}]\ \texttt{body}_{\texttt{inner}})]\!] \Rightarrow \mathcal{T}(nest \triangleright (f(\texttt{flag}) \times : \texttt{inner}\ [: \texttt{guard}\ (\texttt{flag},\ \texttt{sel})]), \texttt{body}_{\texttt{inner}})$$

$$\mathcal{S}(nest)[\![(\texttt{v} \leftarrow (\texttt{map exp}_{\texttt{f}}\ \texttt{ls}))]\!] \Rightarrow nest \triangleright (f(\texttt{v}) \times : \texttt{body}\ [\texttt{v}_{\texttt{elem}} = (\texttt{exp}_{\texttt{f}}\ \texttt{ls}_{\texttt{elem}})])$$

$$\mathcal{S}(nest)[\![(\texttt{v} \leftarrow (\texttt{pack sel ls}_{\texttt{elem}}))]\!] \Rightarrow nest \triangleright (f(\texttt{sel}) \times : \texttt{body}\ [\texttt{v}_{\texttt{elem}} = \texttt{ls}_{\texttt{elem}}]$$

$$\begin{aligned}\mathcal{S}(nest)[\![(\texttt{n} \Leftarrow (\texttt{reduce exp}_{\texttt{f}}\ \texttt{v}_{\texttt{init}}\ \texttt{ls}))]\!] \Rightarrow nest &\triangleright (\top \times : \texttt{start}\ [\texttt{acc} = \texttt{v}_{\texttt{init}}])\\ &\triangleright (f(\texttt{ls}) \times : \texttt{body}\ [\texttt{acc} = \texttt{exp}_{\texttt{f}}\ \texttt{acc}\ \texttt{ls}_{\texttt{elem}}\ ])\\ &\triangleright (\top \times : \texttt{end}\ [(\texttt{read acc})])\end{aligned}$$

$$\begin{aligned}\mathcal{S}(nest)[\![(\texttt{v} \leftarrow (\texttt{create ls}))]\!] \Rightarrow nest &\triangleright (\top \times : \texttt{start}\ [\texttt{v} = \texttt{newVec}\ k])\\ &\triangleright (f(\texttt{v}) \times : \texttt{body}\ [\texttt{v}_{\texttt{elem}} = \texttt{ls}_{\texttt{elem}}])\\ &\triangleright (\top \times : \texttt{end}\ [(\texttt{slice}\ k\ \texttt{vec})])\end{aligned}$$

Figure 4. Scheduling abstract loop nests form

Table I
CONCRETIZING ALNF TO JAVA CODE

Attribute	Abstract loop nests form (k : rate variable)	Generated Java code
create new Vector	$\texttt{vec} = \texttt{newVec}\ k$	$\texttt{vec} = \texttt{new Object}[\texttt{size}_k];$
create new Variable	$\texttt{var} = \texttt{newAcc exp}_{\texttt{init}}$	$\texttt{var} = \texttt{exp}_{\texttt{init}};$
iterator	$\texttt{var} = \texttt{next}\ k\ \texttt{var}_{\texttt{seq}}$	$\texttt{var} = \texttt{var}_{\texttt{seq}}[k];$
assign (function call)	$\texttt{var} = \texttt{f ls}\ \cdots$	$\texttt{var} = \texttt{f.invoke}(\texttt{ls.nth}(k),\ \cdots);$
assign (rename)	$\texttt{var} = \texttt{var}'$	$\texttt{var} = \texttt{var}';$
assign (accumrate)	$\texttt{acc} = \texttt{f acc ls}$	$\texttt{acc} = \texttt{f.invoke}(\texttt{acc},\ \texttt{ls.nth}(k));$
write to a vector	$\texttt{writeVec}\ k\ \texttt{vec ls}_{\texttt{elem}}$	$\texttt{vec}[k] = \texttt{ls}_{\texttt{elem}};$
read variable	$(\texttt{read acc})$	$-$
slice vector	$(\texttt{slice}\ k\ \texttt{vec})$	$-$
Loop frame	$: \texttt{loop}\ (k)$	$\texttt{for}\ (k = 0;\ k < \texttt{size}_k; k++)$ $\{\ \texttt{body}_{\texttt{inner}}\ \}$
Guard frame	$: \texttt{guard}\ (\texttt{flag},\ \texttt{sel})\ {}^*(k = f(\texttt{sel}))$	$\texttt{if}\ (\texttt{flag})\ \{\ \texttt{body}_{\texttt{inner}}\ k{+}{+};\}$

flow fusion can. That code appears in the situation to find the maximum value, the minimum value, the summation or specific values of the single list at the same time.

In general, data flow fusion may fuse other cases. The simple example to reduce two vectors of the same length is the evaluation of Pearson correlation coefficient r of the two vectors $\mathbf{x} = (x_1, \cdots, x_n), \mathbf{y} = (y_1, \cdots, y_n)$ in statistics. The r is defined as

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}.$$

After the calculation of the averages, $\bar{x}$ and $\bar{y}$, the evaluation of r is able to express as single loop, on the ground that the two vectors are the same length n and that code can consist of map, filter and reduce.

VI. BENCHMARK TESTS

To evaluate the effectiveness of our macro, we do benchmark tests of four kinds of program, map-map-reduce, Pearson correlation coefficient (PCC), multiple reduces, and filter sum. Map-map-reduce is the simple fusion of two maps and a reduce. PCC and multiple reudces are described as in the section V. filter sum is to extract data with a predicate and take that summary.

We examine our macro in the point of the elapsed times for three kinds of program style, unfused, hand-fused, and macro-used. Hand-fused codes is fused with loop-recur special form. The test executed on Windows 8.1 Enterprise with Intel Core i5-4300U CPU @ 1.90GB with 8GB of RAM.

The results of benchmark tests are in Table II. The times of benchmark are not stable, so Table II is an example. The unit of time is milliseconds.

Table II
BENCHMARKS

-	Unfused	Hand-fused	Macro-used
Map-map-reduce	2.74	5.69	2.22
PCC	104.00	107.59	100.96
Multiple reduces	2.55	5.93	2.77
Filter sum	3.28	-	2.51

```
import clojure.lang.APersistentVector;
import clojure.lang.IFn;
public class FlowFusionGenerated2675{
  public static APersistentVector vec1;
  public static IFn fn2676;
  public static IFn fn2677;
  public static IFn fn2678;
  public static Object n2659;
  public static Object[] new_lseq2660;
  public static void loop(){
    int k2672 = 0; int k2674 = 0;
    int k2673 = 0; int k2671 = 0;
    n2659 = 0;
    new_lseq2660 = new Object [vec1.length()];
    int size2679 = vec1.length();
    for (k2673 = 0; k2673<size2679; k2673++){
      Object vec22655 =
        fn2676.invoke(vec1.nth(k2673));
      boolean flags2656 =
        (boolean)fn2677.invoke(vec22655);
      if (flags2656) {
        Object vec32658 = vec22655;
        n2659 = fn2678.invoke(n2659, vec32658);
        new_lseq2660[k2672] = vec32658 ;
        k2672++;
      }}}}
```

Figure 6. Generated Java code for filter max

VII. Discussion

The macro-used codes are faster than the unfused code except for multiple reduces, but the other values are less effective in spite of the retrieval of intermediate data structures and unnecessary loops. Besides, it's better not to do the hand-fusion with loop-recur form in Clojure.

There are three reasons of the instability of the benchmark tests. At first, our macro can't make the target program faster in this benchmark tests. The test programs might be affected by the optimizations in the JVM. The typical optimization is JIT compilation. Therefore, benchmark tests on JVM is difficult in general [11]. Secondly, it's considered the effect of the garbage collector. The last cause is the effects of an interpretation on Clojure. These are the lazy sequence, the reflections of JVM, dynamic typing mechanism, and the garbage collection of Clojure's objects.

It is difficult to find the best code in the point of speed. The benchmark test might be also trapped by those several reasons.

A way of the decision whether the macro make code faster than the previous code is to implement on other languages, such as Scheme, LFE (Lisp Flavored Erlang), Hy, or Common Lisp not on the JVM.

Moreover, there is future work except for the speeds. It's to add another functions of list operations. Clojure has many sequential (list) operations. For example, These are interpose/interleave functions that interpose the elements into the target list. The loop rates of those functions are decidable.

VIII. Conclusion

We introduce a optimization technique for a Clojure code with the macro and implement data flow fusion for dynamic typing mechanism.

The transformation technique has three principal transformations. The first step is from the DSL code to A normal form, which is a popular method for compilation from a functional programming language to an imperative programming language. In this step, we introduce the type inference for dynamic typed mechanism and add create functions following the results of type inference.

And second, our macro compiles an A-Normal form to DFPD, for readjusting the code and compiles from DFPD to ALNF. While compiling to ALNF, the loops in the program have been fused each other. In this step, we follow Lippmeier's technique [7].

Finally, we implement the generator of the concrete Java code and Clojure code. The codes derive from the ALNF and are compiled to a class file of Java and an S-expression. A standard Java compiler compiles generated Java code, and Clojure's macro processor embeds the generated Clojure code in the whole code.

Our macro has many applicable situations, because a composition of map, filter and reduce generates several programs and data flow fusion can fuse the code containing several list consumers for one list. Additionally, programmers can embed the macro in a general program of Clojure.

In the benchmark tests, we conclude that our macro makes code faster than the previous code in some cases. However, the macro is not always effective in the cases of the calculation PCC and the multiple reduces. A benchmark test on the JVM sometimes causes unstable elapsed times of the code.

The future works are the precision of the benchmark tests on JVM and to add other list operations and special forms of Clojure.

References

[1] Clojure, *http://clojure.org/*

[2] Clojure - Transducers, *http://clojure.org/reference/transducers*

[3] A. Gill, J. Launchbury, S. Jones, A Short Cut to Deforestation, *Conference on Functional Programming Languages and Computer Architecture*, 1993.

[4] E. Meijer, M. Fokkinga, R. Paterson. Functional pro- gramming with bananas, lenses, envelopes and barbed wire, *In FPCA, Functional Programming Languages and Computer Architecture.* ACM, 1991.

[5] Y. Onoue, Z. Hu, H. Iwasaki, M. Takeichi, A Calculational Fusion System HYLO, *In IFIP, TC2 Working Conference on Algorithmic Languages and Calculi, Chapman and Hall*, 1997.

[6] D. Coutts, R. Leshchinskiy, D. Stewart, Stream fusion: from lists to streams to nothing at all, *International Conference on Functional Programming*, ACM, 2007

[7] B. Lippmeier, M. M. T. Chakravarty, G. Keller, A. Robinson, Data flow fusion with series expression in Haskell, *ACM SIGPLAN symposium on Haskell*, 2013.

[8] S. Chatterjee, G. E. Blelloch, and A. L. Fisher. Size and access inference for data-parallel programs, *In PLDI, Programming Language Design and Implementation.* ACM, 1991.

[9] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen, The Essence of Compiling with Continuations, *In PLDI, Programming Language Design and Implementation.* ACM, 1993.

[10] J. Gibbsons and O. Moor, the fun of programming Palgrave Macmillan, 2005.

[11] S. Oaks, Java Performance: The Definitive Guide, O'Reilly Media, Inc, 2014.