# A Case Study on a Modular Transformation Strategy

Zhenjiang Hu  Wei-Ngan Chin  Masato Takeichi

Technical Report METR 99-06
Department of Mathematical Engineering
University of Tokyo

**Summary.**
Transformational programming is a well-known methodology to derive both correct and efficient programs. But it often requires deep insights to make major jumps during derivation, and so it remains unclear how general a derivation for one problem can be applied to others, particularly to those whose efficient algorithms are unknown. In this paper, we show that it is possible to minimize these deep insights. Our thesis is that the high-level transformation techniques such as fusion, tupling, and generalization/accumulation can be well integrated to help provide a systematic and modular approach to calculate efficient programs, and thus the mild insights in our transformation are mainly confined to meet the conditions to facilitate transformation techniques. We illustrate our approach by a case study on the derivation of a new efficient algorithm for finding frequent sets, one of the basic building blocks of many data mining algorithms.

**Keywords**: Program Transformation, Functional Programming, Bird Meertens Formalisms, Frequent Set Problem.

## 1 Introduction

When writing a program, the programmer is faced with a tension between correctness and efficiency. A program that is easy to understand and whose correctness is obvious to see often fails to be efficient, while a more efficient program often compromises clarity . *Transformational programming* [BD77, Fea87, Dar81, Bir84, Bir86, Bir87, Bac95] is a well known methodology to address this difficulty.

In transformational programming, one does not attempt to produce directly a program that is correct, understandable and efficient, rather one initially concentrates on producing a program which is as clear and understandable as possible ignoring any question of efficiency. Having satisfied himself that he has a correct program he successively transforms it to more and more efficient versions using methods guaranteed to preserve the meaning of the program.

Although quite a lot of creative, elegant and efficient algorithms [Bir84, Bir86, PP96] have been derived in this manner showing the impetus of the transformational approach to programming, there remain two major problems.

- *Insightful rules for big-step jumps can be difficult to find.* While creative algorithms are interesting to exhibit, they often require deep insights to make major jumps to the transformed code. This can make things very difficult both for human to comprehend and for machine to implement.
- *Application scope is unclear.* It is usually not so clear how a derivation for obtaining efficient programs for a specific problem can be applied generally to others, particularly to those problems whose efficient algorithms are still unknown.

The main purpose of this paper is to show that it is possible to minimize these deep insights. Our thesis is that the high-level transformation techniques [Fea87, PP96] such as fusion, tupling, and generalization/accumulation can be well integrated to help provide a systematic and modular approach to calculate efficient programs, and thus the mild insights in our transformation are mainly confined to meet the condition to facilitate transformation techniques.

To appreciate the virtues of our modular approach to transformation, we focus on a case study of the problem for finding frequent sets (see Section 4.1), one of the basic building blocks of many data mining algorithms [AIS93, MT96]. Suppose that an organization has recorded the set of objects purchased by each customer on each visit. The goal of the frequent set problem is to find those subsets of objects that appear frequently in customers' visits. This information can be used to, for example, place objects that are often purchased together near each other on the shelf. The motivations for choosing the frequent set problem as our case study are two folds.

- First, and most importantly, the frequent set problem is a practical problem that is fundamental in data mining, where a widely accepted efficient algorithm is still unknown. We need a clever algorithm in practice, because the size of the data concerned is so great that it becomes critical to do this task as efficiently as possible.
- Second, the frequent set problem fits well with transformational approaches in the sense that it is straightforward to write down a correct solution (Sect. 4.1), but is far from trivial to transform it to an efficient solution. To the best of our knowledge, no such attempt has been reported yet.

In this paper, we shall illustrate our modular transformation strategy via this case study. Our main contributions can be summarized as follows.

- We propose a *modular* transformation that supports the reuse of codes and transformation techniques. The basis of our approach is the identification of a small set of commonly used transformation techniques. Particularly, we highlight three important transformation techniques, namely fusion, tupling and accumulation, which can be well combined together for calculating efficient programs.
- Our transformation is more *systematic*, minimizing the use of complex laws with deeper insights, such as Horner's rule used in [Bir87], which tend to make transformation harder to carry out. Instead, our approach relies on a

set of smaller laws which are motivated by the need to perform transformation techniques.

- Our modular transformation is *powerful*. To the best of our knowledge, we have calculated a new algorithm for finding frequent sets, which is much faster and more compact than hand-coded programs in Haskell for some existing algorithms.

This paper is organized as follows. In Sect. 2, we review the notational conventions and some basic concepts used in this paper, and outline several useful transformation techniques. After explaining briefly our modular approach to transformation in Sect. 3, we focus on the frequent set problem, and show how to apply the modular transformation approach to derive a new efficient algorithm in Sect. 4. Related work and concluding remarks are give in Sect. 5.

## 2 Preliminaries

In this section, we briefly review the notational conventions and some basic concepts of Bird Meertens Formalisms (BMF for short) [Bir87], and outline several useful transformation techniques which will be used in the rest of this paper.

### 2.1 BMF

We use BMF, a program calculus, to describe our transformation. Those who are familiar with functional languages like Haskell should have no difficulty in understanding its notational conventions.

**Functions**

*Function application* is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function composition is an associative operator, and the identity function is denoted by $id$. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*: an infix binary operator like $\oplus$ can be turned into unary functions by

$$(a\oplus)\,b = a \oplus b = (\oplus\,b)\,a.$$

In addition, we can turn prefix binary operators into infix ones by back-quotation as follows.

$$f\ x\ y = x\ \text{`}f\text{`}\ y$$

**Lists**

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element $a$ (and $[\cdot]$ for the function taking $a$ to $[a]$), and $x +\!\!+\, y$ for the concatenation of two lists $x$ and $y$. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : xs$ for $[a] +\!\!+\, xs$.

### Higher Order Functions: Map and Filter

BMF has many useful higher order functions which enjoy very nice algebraic properties. Among which, the *map* and *filter* will be used in this paper.

Map is the operator which applies a function to every element in a list. It is written as an infix $*$, formally defined by

$$\begin{array}{lcl} f * [\,] & = & [\,] \\ f * (x : xs) & = & f\ x\ :\ f * xs. \end{array}$$

It satisfies the so-called *map-distributivity* property:

$$f * \ \circ \ g* = (f \circ g) * .$$

Filter is the operator which takes a predicate $p$ and a list and return the sublist whose elements satisfy $p$. It is written by $p \triangleleft xs$, which is formally defined by

$$\begin{array}{lcl} p \triangleleft [\,] & = & [\,] \\ p \triangleleft (x : xs) & = & \text{if } p\ x \text{ then } x : p \triangleleft\ xs \text{ else } p \triangleleft\ xs \end{array}$$

The filter enjoys the *filter-element-map* property which is often used in program derivation (e.g., in [Bir84]):

$$(p\triangleleft) \circ ((x\ :)*) = (x\ :) * \circ((p \circ (x\ :))\triangleleft)$$

and the *filter-pipeline* property:

$$p\triangleleft \ \circ \ q\triangleleft = (\lambda x.(p\ x\ \wedge\ q\ x))\triangleleft .$$

## 2.2 Useful Transformation Techniques

During transformation process, we need *transformation techniques* [Fea87] or *strategies* [PP96] that guide the application of the transformation rules and may allow us to derive programs with improved performance. We outline some transformation techniques used in this paper.

- *Fusion* [Wad89, Chi92, OHIT97]. Fusion is to merge nested compositions of functions in order to obtain new functions without unnecessary intermediate data structures. For instance, the composition of function # for computing the length of a list and $(1+)*$ for incrementing every element of a list can be fused into a single # operation as follows:

  $$\# \circ (1+) * \ \Rightarrow\ \#$$

  where the intermediate data structure produced by $(1+)*$ is completely eliminated.

- *Accumulation/Generalization* [Bir84, Pet87, PS87, HIT99]. Accumulation is to generalize a function by inclusion of an extra parameter, called an *accumulating parameter*, for reusing and propagating intermediate results. As an example, consider the following definition of *isum* which computes the initial prefix sums of a list, i.e., $isum\ [x_1, x_2, \cdots, x_n] \ = \ [0, x_1, x_1 + x_2, \cdots, x_1 + x_2 + \cdots + x_n]$:

  $$\begin{array}{lcl} isum\ [] & = & [0] \\ isum\ (x : xs) & = & 0 : (x+) * (isum\ xs). \end{array}$$

It can be transformed into the following by accumulation transformation.

$$\begin{aligned}
isum\ xs &= isum'\ xs\ 0 \\
isum'\ [\,]\ d &= [d] \\
isum'\ (x : xs)\ d &= d : isum'\ xs\ (d + x)
\end{aligned}$$

Here the second parameter of $isum'$ is the accumulating one that keeps partial sums for the later reuse, leading to a more efficient algorithm.

- *Tupling* [Chi93, HITT97]. Tupling is to obtain new efficient recursive functions by grouping some recursive functions manipulating the same data into a tuple or even a table. For example, we can apply the tupling transformation to the Fibonacci function defined by

  $$fib\ n = \text{if } n < 2 \text{ then } n \text{ else } \underline{fib\ (n-1)} + \underline{fib\ (n-2)}$$

  introducing a new function *tup* for grouping the above two underlined parts

  $$tup\ n = (fib\ (n-1), fib\ (n-2)).$$

  Through tupling, an efficient linear program can be derived by induction on $n$.

## 3 A Modular Transformation Approach

We consider programs that can be specified by compositions of functions:

$$prog = pass_m \circ \cdots \circ pass_1.$$

To make it efficient, we combine the useful programming techniques in Section 2.2, and proceed with our derivation using the following three steps.

1. Apply the fusion transformation technique to merge multiple passes into a single one. This may result in a (mutual) recursive definition in the form like:

   $$\begin{aligned}
   f_1\ [\,] &= e_1 \\
   f_1\ (x : xs) &= C_1[f_1\ xs, \ldots, f_n\ xs] \\
   &\ \vdots \\
   f_n\ [\,] &= e_n \\
   f_n\ (x : xs) &= C_n[f_1\ xs, \ldots, f_n\ xs]
   \end{aligned}$$

   Where $C_1, \ldots, C_n$ denote expression contexts. Note that we may introduce some new functions to meet the fusible conditions [HIT97].

2. Apply the generalization/accumulation transformation technique to make use of intermediate results, and to structure the recursive definition. This may result in a new recursive definition with additional accumulating parameter something like:

   $$\begin{aligned}
   f_1\ [\,]\ c &= e_1 \\
   f_1\ (x : xs)\ c &= C_1[f_1\ xs\ c_{11}, f_1\ xs\ c_{12}, \ldots, f_n\ xs\ c_{1m_1}] \\
   &\ \vdots \\
   f_n\ [\,]\ c &= e_n \\
   f_n\ (x : xs)\ c &= C_n[f_1\ xs\ c_{n1}, f_1\ xs\ c_{n2}, \ldots, f_n\ xs\ c_{nm_n}].
   \end{aligned}$$

3. Apply the tupling transformation technique to remove multiple traversals of the data $xs$ by the same or different functions, aiming to derive a linear recursive definition (only a single recursive call appears in the definition body).

$$\begin{aligned} f\ []\ c &= e \\ f\ (x : xs)\ c &= C[f\ xs\ c'] \end{aligned}$$

It is worth noting that although we shall only show how to improve the top-level functions, we can apply this modular transformation strategy to improve functions that are used inside the expression contexts.

## 4 Case Study: Frequent Set Problem

Let us return to our case study: the frequent set problem. We shall start out with a specification, a straightforward program to solve the problem, and then we improve it by means of our modular program transformation. We include some experimental results at the end of this section.

### 4.1 Specification

Suppose that a shop has recorded the set of objects purchased by each customer on each visit. The frequent set problem is to find all subsets of objects that appear frequently in customers' visits with respect to a specific threshold. As an example, suppose a shop has the following object set:

$$\{A, B, C, D, E, F, G, H, I, J, K\}$$

and the shop recorded the following customers' visits:

| | |
|---|---|
| visit 1: | $\{A, B, C, D, G\}$ |
| visit 2: | $\{A, B, E, F\}$ |
| visit 3: | $\{B, I\}$ |
| visit 4: | $\{A, B, H\}$ |
| visit 5: | $\{E, G\}$ |

We can see that $A$ and $B$ appear together in three out of the five visits. Therefore we say that the subset $\{A, B\}$ has frequency ratio of 0.6. If we set the frequency ratio threshold to be 0.3, then we know that the sets of

$$\{A\},\ \{B\},\ \{E\},\ \{G\} \text{ and } \{A, B\}$$

pass this threshold, and thus they should be returned as the result of our frequent set computation.

To simplify our presentation, we impose some assumption on the three inputs, namely object set $os$, customers' visits $vss$, and threshold $least$. We shall represent the objects of interest using an ordered list of integers without duplicated elements, e.g.,

$$os = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]$$

and represent customers' purchasing visits by a sublist of its sublist, e.g.,

$$vss = [[10, 20, 30, 40, 70], [10, 20, 50, 60], [20, 90], [10, 20, 80], [50, 70]].$$

Furthermore, for threshold, we will use an integer, e.g.,

$$least = 3$$

to denote the *least* number of appearances in the customers' visits, rather than using a ratio of the number of appearances against that of the whole visits.

Now we can solve the frequent set problem straightforwardly by the following pseudo Haskell program:

$$\begin{array}{lcl} fs & : & [Int] \to [[Int]] \to Int \to [[Int]] \\ fs\ os\ vss\ least & = & (fsp\ vss\ least) \triangleleft (subs\ os) \end{array}$$

It consists of two passes that can be read as follows.

1. First, we use *subs* to enumerate all the sublists of the object list *os*, where *subs* can be defined by

$$\begin{array}{lcl} subs & : & [a] \to [[a]] \\ subs\ [\,] & = & [[\,]] \\ subs\ (x : xs) & = & subs\ xs +\!\!+ (x :) * subs\ xs. \end{array}$$

2. Then, we use the predicate *fsp* to filter the generated sublists to those that appear frequently (passing the threshold *least*) in customers' visits *vss*. Such *fsp* can be easily defined by

$$\begin{array}{lcl} fsp & : & [[Int]] \to Int \to [Int] \to Bool \\ fsp\ xss\ least\ ys & = & \#((ys\ `isSublist`) \triangleleft xss) \geq least \end{array}$$

Note that $xs$ `isSublist` $ys$, is for deciding if $xs$ is a sublist of $ys$. This operation is defined by:

$$\begin{array}{lcl} [\,]\ `isSublist`\ ys & = & True \\ (x : xs)\ `isSublist`\ ys & = & x\ `isElem`\ ys\ \wedge\ xs\ `isSublist`\ ys \end{array}$$

where $x$ `isElem` $ys$ returns *True* if $x$ is an element of list $ys$.

Being straightforward, this initial program is obviously infeasible for all but the very small object set, because the search space of potential frequent sets consists of $2^{\#os}$ sublists.

## 4.2 Derivation via Modular Transformation

We shall demonstrate how the exponential search space of our initial concise program can be reduced dramatically via our modular transformation in Section 3. That is, we will derive an *efficient* program for finding frequent sets, by applying transformation techniques of fusion, generalization/accumulation, followed by tupling.

## Fusion

Fusion is to merge the two passes into a single one. This can be done by the following calculation through induction on $os$.

$$\begin{array}{ll}
& fs\ [\,]\ vss\ least \\
= & \quad \{\ \text{def. of } fs\ \} \\
& (fsp\ vss\ least) \triangleleft (subs\ [\,]) \\
= & \quad \{\ \text{def. of } subs\ \} \\
& (fsp\ vss\ least) \triangleleft [[\,]] \\
= & \quad \{\ \text{def. of } \triangleleft \text{ and } fsp\ \} \\
& \text{if } \#(([\,]\ `isSublist`) \triangleleft vss) \geq least \text{ then } [[\,]] \text{ else } [\,] \\
= & \quad \{\ isSublist\ \} \\
& \text{if } \#((\lambda ys.True) \triangleleft vss) \geq least \text{ then } [[\,]] \text{ else } [\,] \\
= & \quad \{\ \text{simplification}\ \} \\
& \text{if } \#vss \geq least \text{ then } [[\,]] \text{ else } [\,]
\end{array}$$

And

$$\begin{array}{ll}
& fs\ (o:os)\ vss\ least \\
= & \quad \{\ \text{def. of } fs\ \} \\
& (fsp\ vss\ least) \triangleleft (subs\ (o:os)) \\
= & \quad \{\ \text{def. of } subs\ \} \\
& (fsp\ vss\ least) \triangleleft (subs\ os +\!\!+ (o:) * (subs\ os)) \\
= & \quad \{\ \text{def. of } \triangleleft\ \} \\
& (fsp\ vss\ least) \triangleleft (subs\ os) +\!\!+ \\
& (fsp\ vss\ least) \triangleleft ((o:) * (subs\ os)) \\
= & \quad \{\ \text{by } \textit{filter-element-map} \text{ property}\ \} \\
& (fsp\ vss\ least) \triangleleft (subs\ os) +\!\!+ \\
& (o:) * ((fsp\ vss\ least \circ (o:)) \triangleleft (subs\ os)) \\
= & \quad \{\ \text{the calculation below}\ \} \\
& (fsp\ vss\ least) \triangleleft (subs\ os) +\!\!+ \\
& (o:) * ((fsp\ ((o\ `isElem`) \triangleleft vss)\ least) \triangleleft (subs\ os))
\end{array}$$

To complete the above calculation, we need to show that

$$fsp\ vss\ least \circ (o:)\ =\ fsp\ ((o\ `isElem`) \triangleleft vss)\ least.$$

This can be easily shown by the following calculation.

$$\begin{array}{ll}
& fsp\ vss\ least \circ (o:) \\
= & \quad \{\ \text{def. of } fsp\ \} \\
& (\lambda ys.(\#((ys\ `isSublist`) \triangleleft vss) \geq least)) \circ (o:) \\
= & \quad \{\ \text{function composition}\ \} \\
& \lambda ys.(\#(((o:ys)\ `isSublist`) \triangleleft vss) \geq least) \\
= & \quad \{\ \text{def. of } isSublist\ \} \\
& \lambda ys.(\#((\lambda xs.(ys\ `isSublist`\ xs\ \wedge\ o\ `isElem`\ xs)) \triangleleft vss) \geq least) \\
= & \quad \{\ \text{by } \textit{filter-pipeline} \text{ property}\ \} \\
& \lambda ys.(\#((ys\ `isSublist`) \triangleleft ((o\ `isElem`) \triangleleft vss)) \geq least) \\
= & \quad \{\ \text{def. of } fsp\ \} \\
& fsp\ ((o\ `isElem`) \triangleleft vss)\ least
\end{array}$$

To summarize, we have obtained the following program, in which all intermediate results used to connect the two passes have been eliminated.

$$\begin{array}{ll} fs\ [\,]\ vss\ least & = \text{if } \#vss \geq least \text{ then } [[\,]] \text{ else } [\,] \\ fs\ (o : os)\ vss\ least & = fs\ os\ vss\ least\ \mathbin{+\!\!+}\ \underline{(o\,:)*}(fs\ os\ ((o\ \text{`}isElem\text{`}) \triangleleft vss)\ least)) \end{array}$$

## Generalization/Accumulation

Notice that the underlined part in the above program for insert $o$ to all elements of the list is rather expensive. Fortunately, this could be improved by means of introducing an accumulating parameter in much the same spirit as [Bir84, HIT99]. To this end, we generalize $fs$ to $fs'$, by introducing an accumulating parameter as follows.

$$fs'\ os\ vss\ least\ r = (r \mathbin{+\!\!+}) * (fs\ os\ vss\ least)$$

And clearly we have

$$fs\ os\ vss\ least = fs'\ os\ vss\ least\ [\,].$$

Calculating the definition for $fs'$ is easy by induction on $os$, and thus we omit the detailed derivation. The end result is as follows.

$$\begin{array}{ll} fs'\ [\,]\ vss\ least\ r & = \text{if } \#vss \geq least \text{ then } [r] \text{ else } [\,] \\ fs'\ (o : os)\ vss\ least\ r & = fs'\ os\ vss\ least\ r\ \mathbin{+\!\!+} \\ & \quad\ fs'\ os\ ((o\ \text{`}isElem\text{`}) \triangleleft vss)\ least\ (r \mathbin{+\!\!+} [o]) \end{array}$$

The accumulation transformation has successfully turned an expensive map operator of $(o\,:)*$ into a simple operation that appends $o$ to $r$. In addition, we have got a nice side-effect from accumulation transformation that $fs'$ is defined in an *almost* tail recursive form, in the sense that each recursive call produces independent part of the resulting list. This kind of recursive form could help us to discover nice properties of function from its base case definition. This technique is called the *base case filter* in [Chi90]. For the above $fs'$, we can deduce from its base case definition that if

$$\#vss < least$$

then

$$fs'\ os\ vss\ least\ r == [\,].$$

We shall call it the *pruning property* that will be used in the later derivation.

## Tupling (Tabulation)

Although much improvement has been achieved through fusion and accumulation, there still remains a source of serious inefficiency because the main parameter $os$ is traversed multiple times by $fs'$, resembling the case for $fib$ function in Section 2.2. This inefficient state shall be handled by the tupling transformation.

The tupling technique that we shall use is called *dynamic tupling* [CH95] or *tabulation* [Bir80], as opposed to the static tupling where we know statically the fixed number of values that should be memoized. The purpose of our dynamic tupling is to merge two recursive calls of $fs'$ together so that $os$ can be traversed once. The difficulty in such tupling is to determine which values should be tabulated. Now, taking a close look at the derived definition for $fs'$

$$\begin{array}{l} fs'\ (o:os)\ \underline{vss}\ least\ \underline{r} = fs'\ os\ \underline{vss}\ least\ \underline{r}\ \mathbin{+\!\!+} \\ \qquad\qquad\qquad\qquad\qquad\quad fs'\ os\ \underline{((o\ `isElem`) \triangleleft vss)}\ least\ \underline{(r \mathbin{+\!\!+} [o])}) \end{array}$$

reveals some dependency of the second and the fourth arguments of $fs'$ among the left and the right recursive calls to $fs'$, as indicated by the underlined parts. Moreover these two arguments will be used to produce the final result, according to the base case definition of $fs'$. This hints us to keep (memoize) all necessary intermediate results of the second and the fourth parameters in a list like

$$[(r_1, vss_1), (r_2, vss_2), \ldots]$$

where we can suppose that each element $(r_i, vss_i)$ meets the *invariant*

$$\#vss_i \geq least.$$

as hinted by the conditional part in the definition of $fs'$.

Now we apply the tupling transformation to $fs'$ by defining

$$\begin{array}{ll} tup\ os\ least\ [\,] & = [\,] \\ tup\ os\ least\ ((r, vss) : ts) & = fs'\ os\ vss\ least\ r \mathbin{+\!\!+} tup\ os\ least\ ts \end{array}$$

and clearly $fs'$ is a special case of *tup*:

$$fs'\ os\ vss\ least\ r = tup\ os\ least\ [(r, vss)].$$

We hope to synthesize a new definition that defines *tup* inductively on *os* where *os* is traversed only once (it is now traversed by both $fs'$ and *tup*). The general form for this purpose will be

$$\begin{array}{ll} tup\ [\,]\ least\ ts & = select\ least\ ts \\ tup\ (o:os)\ least\ ts & = tup\ os\ least\ (add\ o\ least\ ts) \end{array}$$

where *select* and *add* are two newly introduced functions that are to be transformed further. We can synthesize *select* by induction on *ts*. From

$$\begin{array}{ll} & tup\ [\,]\ least\ [\,] \\ = & \quad \{ \text{def. of } tup \} \\ & [\,] \end{array}$$

and

$$\begin{array}{ll} & tup\ [\,]\ least\ ((r, vss) : ts) \\ = & \quad \{ \text{relation between } tup \text{ and } fs'\text{, the invariant tells: } \#vss \geq least\ \} \\ & fs'\ [\,]\ vss\ least\ r \mathbin{+\!\!+} tup\ [\,]\ least\ ts \\ = & \quad \{ \text{def. of } fs' \} \\ & (\text{if } \#vss \geq least \text{ then } [r] \text{ else } [\,]) \mathbin{+\!\!+} tup\ [\,]\ least\ ts \\ = & \quad \{ \text{ by the invariant: } \#vss \geq least\ \} \\ & [r]\ \mathbin{+\!\!+}\ tup\ [\,]\ least\ ts \\ = & \quad \{ \text{relation between } tup \text{ and } select \} \\ & [r]\ \mathbin{+\!\!+}\ select\ least\ ts \end{array}$$

we soon have

$$\begin{array}{ll} \mathit{select\ least}\ [\,] & = [\,] \\ \mathit{select\ least}\ ((r, vss) : ts) & = [r]\ +\!\!+\ \mathit{select\ least\ ts} \end{array}$$

i.e.,

$$\mathit{select\ least} = fst * .$$

The definition of *add* can be inferred in a similar fashion. And we can obtain

$$\begin{array}{ll} \mathit{add\ o\ least}\ [\,] & = [\,] \\ \mathit{add\ o\ least}\ ((r, vss) : ts) & = (\text{if}\ \#((o\ `\mathit{isElem}`) \triangleleft vss) < \mathit{least} \\ & \quad\ \text{then}\ [(r, vss)] \\ & \quad\ \text{else}\ [(r, vss), (r +\!\!+ [o], (o\ `\mathit{isElem}`) \triangleleft vss)]) +\!\!+ \\ & \quad \mathit{add\ o\ least\ ts}. \end{array}$$

Notice the crucial calculation step of using the pruning property in the derivation of *add*. It plays an important role in elimination of unnecessary computation.

**A Further Improvement**

The algorithm we obtained is very efficient; it does not contain unnecessary intermediate result, only needs a single pass of the objects of interest, and make use of some necessary intermediate result. Further improvement can be made to improve the functions that are used in the definition body. In fact, we can *specialize* the general *isElem* to the following more efficient one using our assumption that all sublists are in an increasing order.

$$\begin{array}{ll} \mathit{isElem} & : \mathit{Int} \to [\mathit{Int}] \to \mathit{Bool} \\ e\ `\mathit{isElem}`\ [\,] & = \mathit{False} \\ e\ `\mathit{isElem}`\ (x : xs) & = \text{if}\ e < x\ \text{then}\ \mathit{False} \\ & \quad\ \text{else if}\ e == x\ \text{then}\ \mathit{True} \\ & \quad\ \text{else}\ e\ `\mathit{isElem}`\ xs \end{array}$$

Putting all together, we get the final result in Fig. 1.

## 4.3 Experiment

So far, we have successfully reached a new algorithm for finding frequent sets. To see how efficient our algorithm is, we will not give a formal study of the cost which needs to take account of the distribution in addition to the size of data. Rather we use a *simple* experiment to compare our algorithm with the existing algorithm[1] [MT96] that is quite commonly used in the data mining community.

We tested the following three programs, whose source codes can be found in the appendix.

- Program 1: our initial program
- Program 2: a functional coding of an existing algorithm [MT96]
- Program 3: our final program

[1] The idea of the algorithm is to generate all frequent sets of length $l + 1$ from those of length $l$, starting from the frequent sets of length 1.

```
fs os vss least              = fs' os vss least []
fs' os vss least r           = tup os least [(r, vss)]
tup [] least ts              = select least ts
tup (o : os) least ts        = tup os least (add o least ts)
select least                = fst*
add o least []               = []
add o least ((r, vss) : ts)  = (if #vss' < least
                                 then [(r, vss)]
                                 else [(r, vss), (r ++ [o], vss')]) ++ add o least ts
                                where vss' = (o `isElem`) ◁ vss
e `isElem` []                = False
e `isElem` (x : xs)          = if e < x then False
                                else (if e == x then True else e `isElem` xs)
```

**Fig. 1** Our Final Program for Finding Frequent Sets

The input sample data was generated randomly. We generated object list of size 20, then generated its sublists of size 100, and set the threshold to be 10 (10% of frequency). We used Glasgow Haskell Compiler together with its profiling mechanism. The experimental result is as follows.

| | total time (secs) | memory cells (mega bytes) |
|---|---|---|
| Program 1 | 1209.06 | 3922.5 |
| Program 2 | 14.3 | 86.5 |
| Program 3 | 0.72 | 1.6 |

It shows that our final program has been dramatically improved comparing to our initial one, and that it is also much more efficient than the functional coding of an existing algorithm (about 20 times faster but using just 1/50 of memory cells).

A fairer comparison may be to implement the three algorithms using traditional languages, say C, and to use a more practical sample data. We are working on this now.

## 5 Related Work and Concluding Remarks

Besides the related work explained in the introduction, we show others below.

Some attempts have been made to combine the existing transformation techniques for program development. Bird [Bir84] showed how to derive a more efficient algorithm by combining promotion (sort of fusion) with accumulation techniques. Chin [Chi95] investigated the synergies and conflicts when applying fusion and tupling interleavely. And Hu et. al [HIT97] shows that a more modular approach of combining fusion and tupling is practically useful. In this paper, we extend this further, showing that accumulation can be usefully combined with fusion and tupling.

This work is much related to the study on data mining [AIS93, MT96]. In particular, the excellent thesis [Toi96] gives an extensive study on the frequent set problems. We deliberately avoid surveying the existing algorithms for the frequent set problem before the derivation of our algorithm, in order to see how much we could get from our proposed modular transformation. We are happy that our modular transformation gave a very efficient algorithm that has not appeared in [Toi96]. In addition, our algorithm is provably correct due to our use of correctness-preserving transformation steps. We are working on a survey of other existing algorithms for the frequent set problem.

This work is a continuation of our effort to apply calculational transformation techniques [THT98] to the development of efficient programs [OHIT97, HITT97, HTC98]. Our previous work put emphasis on mechanical implementation of the transformation techniques, while this paper aims to show that modular transformation strategy is also very helpful for guiding programmers/researchers in development of new algorithms.

## Acknowledgments

This paper owes much to the thoughtful and inspiring discussions with David Skillicorn. He first recognized that program calculation might be useful to derive an efficient algorithm to solve the frequent set problem. He kindly explained to the first author the problem as well as some existing algorithms, and generously shared his idea with us during the development of our efficient algorithm. We would also like to thank Christoph Armin Herrmann who gave us his functional coding of an existing algorithm, and helped testing our programs with his HDC system.

## References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *1993 International Conference on Management of Data (SIGMOD'93)*, pages 207–216, May 1993.

[Bac95] R. Backhouse. The calculational method. *Special Issue on the Calculational Method, Information Processing Letters*, 53:121, 1995.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bir80] R. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.

[Bir84] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir86] R. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6(2):159–189, 1986.

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[CH95] W. Chin and M. Hagiya. A transformation method for dynamic-sized tabulation. *Acta Informatica*, 32:93–115, 1995.

[Chi90] W.N. Chin. *Automatic Methods for Program Transformation.* Phd thesis, Department of Computing, Imperial College of Science, Technology and Medicone, University of London, May 1990.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[Chi93] W. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, June 1993. ACM Press.

[Chi95] W. Chin. Fusion and tupling transformations: Synergies and conflits. In *Proc. Fuji International Workshop on Functional and Logic Programming*, pages 106–125, Susono, Japan, July 1995. World Scientific.

[Dar81] J. Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.

[Fea87] M.S. Feather. A survey and classification of some program transformation techniques. In *TC2 IFIP Working Conference on Program Specification and Transformation*, pages 165–195, Bad Tolz, Germany, 1987. North Holland.

[HIT97] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[HIT99] Z. Hu, H. Iwasaki, and M. Takeichi. Caculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[MT96] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 189 – 194, Portland, Oregon, August 1996. AAAI Press.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106, Le Bischenberg, France, February 1997. Chapman&Hall.

[Pet87] A. Pettorossi. Program development using lambda abstraction. In *Int'l Conf. on Fundations of Software Technology and Theoretical Computer Science*, pages 420–434, Pune, India, 1987. Springer Verlag (LNCS 287).

[PP96] A. Pettorossi and M. Proiett. Rules and strategies for transforming functional and logic programs. *Computing Surveys*, 28(2):360–414, June 1996.

[PS87] A. Pettorossi and A. Skowron. Higher-order generalization in program derivation. In *Conf. on Theory and Practice of Software Development*, pages 182–196, Pisa, Italy, 1987. Springer Verlag (LNCS 250).

[THT98] A. Takano, Z. Hu, and M. Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 30(3), December 1998. Special issues for 1998 Symposium on Partial Evaluation.

[Toi96] H. Toivonen. *Discovery of Frequent Patterns in Large Data Collections.* Ph.D thesis, Department of Computer Science, University of Helsinki, 1996.

[Wad89] P. Wadler. Theorems for free. In *Proc. Conference on Functional Programming and Computer Architecture*, pages 347–359, 1989.

**Program 1: our initial straightforward program**

```
fs :: [Int] -> [[Int]] -> Int -> [[Int]]
fs os vss least = filter (fsp vss least) (subs os)

fsp :: [[Int]] -> Int -> [Int] -> Bool
fsp vss least ys = length (filter (ys ‘isSublist‘) vss) >= least

[] ‘isSublist‘ ys = True
(x:xs) ‘isSublist‘ ys = (x ‘isElem‘ ys) && (xs ‘isSublist‘ ys)

e ‘isElem‘ [] = False
e ‘isElem‘ (x:xs) = if e==x then True else e ‘isElem‘ xs

subs [] = [[]]
subs (x:xs) = subs xs ++ (x:)* (subs xs)
```

**Program 2: a program coding an existing algorithm**

```
fs :: [Int] -> [[Int]] -> Int -> [[Int]]
fs os vss least = let ubnd = foldr max 0 (length* vss)
                  in  datamineSet os ubnd vss least

datamineSet :: [Int] -> Int -> [[Int]] -> Int -> [[Int]]
datamineSet os u vss least
 = let -- list of single items that satisfy fraction condition:
       siOK = map (\x->[x]) (filter (\x -> fracOK vss least [x]) os)
   in  (fst (foldl (freqSets vss least (map (!!0) siOK)) (siOK,siOK) [2..u]))

freqSets :: [[Int]] -> Int -> [Int] ->
            ([[Int]],[[Int]]) -> Int -> ([[Int]],[[Int]])
freqSets vss least sngl (f,filast) i
 = let fi = filter (fracOK vss least)
                   (remDuplicates (filter (\xs -> length xs == i)
                                 [ insertSet s1 s2 | s1 <- sngl, s2 <- filast]))
   in  (f++fi,fi)

fracOK :: [[Int]] -> Int -> [Int] -> Bool
fracOK bs least b = countSubsets b bs >= least

compareSet :: Ord a => [a] -> [a] -> Int
compareSet xs ys
 = if length xs == length ys
      then let firstdiff =
                 skel_while (\i -> if (i<length xs) then (xs!!i==ys!!i)
                                                    else False ) (+1) 0
           in if firstdiff == length xs
                 then 0
                 else if xs!!firstdiff > ys!!firstdiff then 1 else (-1)
      else if length xs > length ys then 1 else (-1)
```

```
isElem :: Ord a => a -> [a] -> Bool
isElem e s = any (==e) s

isSubSet :: Ord a => [a] -> [a] -> Bool
isSubSet sub super = all (\s -> isElem s super) sub

insertSet :: Ord a => a -> [a] -> [a]
insertSet x xs = filter (<x) xs ++ (x : filter (>x) xs)

remDuplicates :: Ord a => [[a]] -> [[a]]
remDuplicates
 = let pivot xs = xs!!(length xs `div` 2)
       p xs = length xs < 2
       b xs = xs
       d xs = let less    = filter (\x -> compareSet x (pivot xs) == (-1)) xs
                  greater = filter (\x -> compareSet x (pivot xs) == 1)    xs
              in [less,greater]
       c xs [as,bs] = as ++ (pivot xs : bs)
   in dc0 p b d c

countSubsets :: [Int] -> [[Int]] -> Int
countSubsets b bs = length (filter (isSubSet b) bs)

skel_while p f v = if p v then skel_while p f (f v) else v

dc0 :: (a->Bool)->(a->b)->(a->[a])->(a->[b]->b)->a->b
dc0 p b d c x = r x
  where r x = if p x then b x else c x (map r (d x))
```

## Program 3: our efficient program

```
fs :: [Int] -> [[Int]] -> Int -> [[Int]]
fs os vss least = fs' os vss least []

fs' os vss least r = tup os least [(r,vss)]

tup [] least ts = select ts
tup (o:os) least ts = tup os least (add o least ts)

select :: [([Int],[[Int]])] -> [[Int]]
select = map fst

add o least [] = []
add o least ((r,vss):ts) = let vss' = filter (o `isElem`) vss
                               ts' = add o least ts
                           in  if length vss' < least
                               then (r,vss):ts'
                               else (r,vss):(o:r,vss'):ts'

e `isElem` [] = False
```

```
e ‘isElem‘ (x:xs) = if e<x then False
                   else (if e==x then True else e ‘isElem‘ xs)
```