Tupling Calculation Eliminates Multiple Data Traversals

Zhenjiang Hu
Department of Information Engineering
University of Tokyo

Hideya Iwasaki
Department of Computer Science
Tokyo University of Agriculture and Technology

Masato Takeichi
Department of Information Engineering
University of Tokyo

Akihiko Takano
Advanced Research Laboratory
Hitachi, Ltd.

Abstract

Tupling is a well-known transformation tactic to obtain new efficient recursive functions by grouping some recursive functions into a tuple. It may be applied to eliminate multiple traversals over the common data structure. The major difficulty in tupling transformation is to find what functions are to be tupled and how to transform the tupled function into an efficient one. Previous approaches to tupling transformation are essentially based on fold/unfold transformation. Though general, they suffer from the high cost of keeping track of function calls to avoid infinite unfolding, which prevents them from being used in a compiler.

To remedy this situation, we propose a new method to expose recursive structures in recursive definitions and show how this structural information can be explored for calculating out efficient programs by means of tupling. Our new tupling calculation algorithm can eliminate most of multiple data traversals and is easy to be implemented.

1 Introduction

Tupling [Bir84, Chi93] is a well-known transformation tactic to obtain new efficient recursive functions without *multiple traversals over the common data structure* (or *multiple data traversals* for short), which is achieved by grouping some recursive functions into a tuple. As a typical example, consider the function *deepest*, which finds a list of leaves that are farthest away from the root of a given tree:

$$\begin{array}{lcl} deepest\ (Leaf\ (a)) & = & [a] \\ deepest\ (Node(l,r)) & = & deepest(l),\ depth(l) > depth(r) \\ & = & deepest(l) +\!\!+ deepest(r), \\ & & \qquad depth(l) = depth(r) \\ & = & deepest(r),\ \text{otherwise} \\ depth\ (Leaf\ (a)) & = & 0 \\ depth\ (Node(l,r)) & = & 1 + max(depth(l), depth(r)) \end{array}$$

The infix binary function $+\!\!+$ concatenates two lists and the function *max* gives the maximum of the two arguments. Being concise, this definition is quite inefficient because *deepest* and *depth* traverse over the same input tree, giving many repeated computations in computing the depth of subtrees. It, however, can be improved with tupling transformation by grouping *deepest* and *depth* to a new function (say *dd*), i.e. $dd\ t = (deepest\ t,\ depth\ t)$, giving the following efficient program.

$$\begin{array}{lcl} deepest\ t & = & \textbf{let}\ (u,v) = dd\ t\ \textbf{in}\ u \\ dd\ (Leaf\ (a)) & = & ([a], 0) \\ dd\ (Node(l,r)) & = & (dpl, 1 + dl), \qquad dl > dr \\ & = & (dpl +\!\!+ dpr, 1 + dl),\ dl = dr \\ & = & (dpr, 1 + dr), \qquad \textbf{otherwise} \\ & & \textbf{where}\ (dpl, dl) = dd\,l \\ & & \qquad\quad (dpr, dr) = dd\,r \end{array}$$

The main problem in the tupling transformation is to find what functions are to be tupled and how an efficient definition for the tupled function is derived. Traditional approaches [Pet87, PP91, Chi93] to solving this problem are based on the well-known *fold/unfold* transformations [BD77], using *tupling analysis* to discover an eureka tuple and using fold/unfold transformation to derive an efficient program for the tupled function. This is quite general but comes at price. In the fold/unfold transformation, it has to keep track of function calls and to use clever control to avoid infinite unfolding. This process introduces substantial cost and complexity, which is actually prevented from being implemented in a real compiler of functional languages.

To remedy this situation, we turn to another transformation technique known as *program calculation* [MFP91, SF93, MH95], which is based on the theory of *Constructive Algorithmics* [Fok92]. Different from the previous fold/unfold transformation whose emphasis is on the generality of transformation process, program calculation deals with programs in some specific recursive forms, such as *catamorphism*, *anamorphism* and *hylomorphism* [MFP91, TM95, HIT96b], and performs transformation based on some local calculational laws. Because of its simplicity of transformation process, program calculation turns out to be easier to be implemented.

This work is greatly inspired by the success of applying the program calculation technique to the fusion transformation [GLJ93, LS95, TM95, HIT96b]. We would like to explore a further possibility to apply the technique of program calculation to tupling transformation, which to the best of our knowledge has never been examined. We are interested in this exploration for two reasons. First, we believe that tupling transformation tactic should be more practical to be used in a compiler. Second, since tupling and fusion are two most related transformation tactics [Chi95], it is quite

2nd ACM SIGPLAN International Conference on Functional Programming, Amsterdam, June 9–11, 1997.

natural to study tupling transformation in the framework where fusion transformation is studied.

In this paper, we demonstrate how to proceed tupling transformation by means of program calculation. Our main contributions are as follows.

- First, we propose a new tupling algorithm to remove multiple data traversals in a program. It is applicable to any lazy functional program. Two important features of our tupling algorithm are:
  - We identify a class of functions called *tuplable functions* which are potentially suitable to be tupled with other functions and enjoy many useful transformation rules for tupling calculation.
  - We calculate an efficient definition for the tupled function in a rather cheap and mechanical way rather than by the expensive fold/unfold transformation,
- Second, our tupling algorithm is given in calculational forms. Therefore, our tupling algorithm preserves the advantages of transformation in calculational forms, as we have seen in the discussion of shortcut deforestation in [GLJ93, TM95]. That is, our algorithm is very general in that it can be applied to recursions over any data structures other than lists, and our algorithm is correct and is guaranteed to terminate.
- Third, our tupling algorithm can coexist well with the shortcut deforestation. Both of them basically rely on the manipulation over catamorphisms. Therefore, it is natural to combine these two techniques in our framework. In contrast, the previous study on this combination based on fold/unfold transformation [Chi95] requires more complicated control to avoid infinite unfoldings in case fusion and tupling are applied simultaneously.

The organization of this paper is as follows. We begin by reviewing in Section 2 the basic concepts of the constructive algorithmics in order to explain the Mutu Tupling Theorem which is the basis of our tupling algorithm. We then propose our main tupling theorem in Section 3. To prove our main theorem, we first investigate on transformation properties of tuplable functions and find how to perform tupling transformation for them in Section 4. And then we give our tupling calculation algorithm in Section 5. Related work and discussion are given in Section 6.

2 Mutu Tupling Theorem

Generally, tupling transformation is very complicated while its termination is far from being trivial [Chi93]. The calculational approach that will be taken here is less general, but should be more practical. We don't guarantee to remove multiple traversals over the same data structures by *all* functions (indeed in a general program we could not hope to do so), but we do allow all legal programs as input and the transformed program will be expected to be made more efficient. Basically, our tupling transformation is based on a single simple rule, the *Mutu Tupling Theorem* [Fok89], which is well-known in the community of *Constructive Algorithmics*.

2.1 Constructive Algorithmics

To understand the Mutu Tupling Theorem, the basis of our tupling algorithm, we should briefly review the previous work [Hag87, MFP91, Fok92] on Constructive Algorithmics, explaining the basic concepts and the notations that will be used in the rest of this paper.

Functors

In Constructive Algorithmics, *polynomial endofunctors* are used to capture the structure of data types, which are built up only by the following four basic functors.

- The **identity** functor I on type X and its operation on functions are defined as follows.

$$I\ X = X, \quad I\ f = f$$

- The **constant** functor $!A$ on type X and its operation on functions are defined as follows.

$$!A\ X = A, \quad !A\ f = id$$

  where id stands for the identity function.
- The **product** $X \times Y$ of two types X and Y and its operation to functions are defined as follows.

$$\begin{array}{lcl} X \times Y & = & \{\,(x,y) \mid x \in X, y \in Y\} \\ (f \times g)\ (x,y) & = & (f\,x,\ g\,y) \\ \pi_1\ (a,b) & = & a \\ \pi_2\ (a,b) & = & b \\ (f\ \triangle\ g)\ a & = & (f\,a,\ g\,a) \end{array}$$

- The **separated sum** $X + Y$ of two types X and Y and its operation to functions are defined as follows.

$$\begin{array}{lcl} X + Y & = & \{1\} \times X \cup \{2\} \times Y \\ (f + g)\ (1,x) & = & (1,\ f\,x) \\ (f + g)\ (2,y) & = & (2,\ g\,y) \\ (f\ \triangledown\ g)\ (1,x) & = & f\,x \\ (f\ \triangledown\ g)\ (2,y) & = & g\ y. \end{array}$$

Although the product and the separated sum are defined over two parameters, they can be naturally extended for n parameters. For example, the separated sum over n parameters can be defined by $\Sigma_{i=1}^{n} X_i = \cup_{i=1}^{n}(\{\mathsf{i}\} \times X_i)$ and $(\Sigma_{i=1}^{n} f_i)\,(\mathsf{j}, x) = (\mathsf{j},\ f_j\ x)$ for $1 \leq \mathsf{j} \leq n$.

Data Types

Rather than being involved in theoretical study which can be found in [Hag87, MFP91, Fok92], we illustrate by some examples how data types can be captured by endofunctors. In fact, from a common data type definition, an endofunctor can be automatically derived to capture its structure [SF93]. As a concrete example, consider the data type of *cons lists* with elements of type A, which is usually defined by[1]

$$List\ A = Nil \mid Cons(A,\ List\ A).$$

[1] Note that for notational convenience, we sometimes use [ ] for Nil and infix operator : for $Cons$. Thus, for example, $x : xs$ stands for $Cons(x, xs)$ and $[a]$ for $Cons(a, Nil)$.

In our framework, we shall use the following endofunctor to describe its recursive structure:

$$F_{L_A} = !\mathbf{1} + !A \times I$$

where $\mathbf{1}$ denotes the final object, corresponding to $()$[2]. Besides, we use $in_{F_{L_A}}$ to denote the *data constructor* in *List* A:

$$in_{F_{L_A}} = Nil \triangledown Cons.$$

In fact, the *List* A is the least solution of X to the equation $X = in_{F_{L_A}}\ (F_{L_A}\ X)$ as discussed in [Hag87]. The $in_{F_{L_A}}$ has its inverse, denoted by $out_{F_{L_A}} : List\ A \to F_{L_A}\ (List\ A)$, which captures the *data destructor* of *List* A, i.e.,

$$\begin{aligned} out_{F_{L_A}} = \lambda xs.\ \text{case } xs \text{ of} \\ Nil \ &\to (1, ()); \\ Cons\ (a, as) &\to (2, (a, as)). \end{aligned}$$

Another example is the data type of *binary trees* with leaves of type A usually defined by

$$Tree\ A = Leaf\ A \mid Node\ (Tree\ A, Tree\ A).$$

The corresponding functor F_{T_A} and data constructor $in_{F_{T_A}}$ are:

$$F_{T_A} = !A\ +\ I \times I, \quad in_{F_{T_A}} = Leaf \triangledown Node.$$

Catamorphisms

Catamorphisms [MFP91, SF93], one of the most important concepts in Constructive Algorithmics, form a class of important recursive functions over a given data type. They are the functions that *promote through* the type constructors. For example, for the cons list, given e and $\oplus$, there exists a unique catamorphism, say *cata*, satisfying the following equations.

$$\begin{array}{lcl} cata\ [\,] & = & e \\ cata\ (x : xs) & = & x \oplus (cata\ xs) \end{array}$$

In essence, this solution is a *relabeling*: it replaces every occurrence of $[\,]$ with e and every occurrence of $:$ with $\oplus$ in the cons list. Because of the uniqueness property of catamorphisms (i.e., for this example e and $\oplus$ uniquely determines a catamorphism over cons lists), we are likely to use special braces to denote this catamorphism as $cata = (\!|e\ \triangledown\ \oplus|\!)_{F_{L_A}}$. In general, a catamorphism over any data type captured by functor F is characterized by:

$$h = (\!|\phi|\!)_F \quad \equiv \quad h \circ in_F = \phi \circ Fh.$$

With catamorphisms many functions can be defined. For example, the function *sum*, which sums up all the elements in a list, can be defined as $(\!|0 \triangledown plus|\!)$[3].

Catamorphisms are *efficient* and *manipulable*: they are efficient in the sense that they traverse over the input data structure once; they are manipulable because they enjoy many useful transformation properties [MFP91, SF93]. To help readers to get used to the notation in this paper, we demonstrate how to inline $sum = (\!|0 \triangledown plus|\!)$ into a familiar program by the following calculation.

$$\begin{array}{ll} & sum = (\!|0 \triangledown plus|\!) \\ \equiv & \quad \{\ \text{catamorphism charaterization}\ \} \\ & sum \circ in_{F_{L_A}} = (0 \triangledown plus) \circ F_{L_A}\, sum \\ \equiv & \quad \{\ in_{F_{L_A}} = (Nil \triangledown Cons),\ F_{L_A} f = id + id \times f\ \} \\ & sum \circ (Nil \triangledown Cons) = (0 \triangledown plus) \circ (id + id \times sum) \\ \equiv & \quad \{\ \text{Laws for } \triangledown, +\text{ and } \circ\ \} \\ & (sum\ Nil) \triangledown (sum \circ Cons) = 0 \triangledown (plus \circ (id \times sum)) \\ \equiv & \quad \{\ \text{by laws of } \triangledown\ \} \\ & sum\ Nil = 0; \quad sum \circ Cons = plus \circ (id \times sum) \end{array}$$

That is,

$$\begin{array}{lcl} sum\ Nil & = & 0 \\ sum\ (Cons(x, xs)) & = & plus(x, sum\ xs). \end{array}$$

2.2 Mutual Recursions and Tupling

Just like that the fusion in calculational forms relies on a single Acid Rain Theorem [TM95], our tupling algorithm basically depends on the Mutu Tupling Theorem [Fok89, Fok92].

Theorem 1 (Mutu Tupling)

$$\frac{f \circ in_F = \phi \circ F(f \vartriangle g),\ g \circ in_F = \psi \circ F(f \vartriangle g)}{f \vartriangle g = (\!|\phi \vartriangle \psi|\!)_F}$$ □

This theorem was originally devised for the purpose of manipulating mutual recursions by turning them into single catamorphisms. However, from the viewpoint of tupling transformation, it provides a simple calculational rule. It not only tells which functions should be tupled — f and g should be tupled if they are mutually defined *traversing over the same data structure* in a *specific regular way*, but also tells how to calculate out the definition for the tupled function — tupling ϕ and ψ within a catamorphism. Therefore, a direct advantage of applying the Mutu Tupling Theorem is that any possible multiple traversals over the same data structure by f and g in the original program can be successfully eliminated, leading to an efficient program. As an example, recall the definition of *deepest* given in the introduction, where *deepest* and *depth* are mutually defined and traverse over the same input tree. We can apply the Mutu Tupling Theorem to obtain an efficient version. First, we rewrite them into the form as required in the theorem:

$$\begin{array}{l} deepest \circ in_{F_{T_{Int}}} = \phi \circ F_{T_{Int}}(deepest \vartriangle depth) \\ \quad \textbf{where}\ \ \phi = \phi_1 \triangledown \phi_2 \\ \qquad \phi_1\ a = [a] \\ \qquad \phi_2\ ((tl, hl), (tr, hr)) = tl, \qquad \textbf{if } hl > hr \\ \qquad\qquad\qquad\qquad\quad\ = tl +\!\!+ tr, \ \textbf{if } hl = hr \\ \qquad\qquad\qquad\qquad\quad\ = tr, \qquad \textbf{otherwise} \end{array}$$

$$\begin{array}{l} depth \circ in_{F_{T_{Int}}} = \psi \circ F_{T_{Int}}(deepest \vartriangle depth) \\ \quad \textbf{where}\ \ \psi = \psi_1 \triangledown \psi_2 \\ \qquad \psi_1\ a = 0 \\ \qquad \psi_2\ ((tl, hl), (tr, hr)) = 1 + max(hl, hr) \end{array}$$

Note that $F_{T_{Int}} f = id + f \times f$ and $in_{F_{T_{Int}}} = Leaf \triangledown Node$. Now applying the Mutu Tupling Theorem soon gives the following efficient linear recursion:

$$deepest = \pi_1 \circ (deepest \vartriangle depth) = \pi_1 \circ (\!|\phi \vartriangle \psi|\!)_{F_{T_{Int}}}$$

which can be inlined to the efficient program as given in the introduction with some simplification.

[2] Strictly speaking, *Nil* should be written as $Nil()$. In this paper, the form of $t\,()$ will be simply denoted as t.

[3] When no ambiguity happens, we usually omit the subscript of F in $(\!|\phi|\!)_F$.

prg	::=	$d_1; \cdots; d_n$	program
d	::=	$eq_1; \cdots; eq_n$	function definition
eq	::=	$f\ p\ v_{s_1}\ \cdots\ v_{s_n} = e$	equation for defining function f
v_s	::=	$v \mid (v_{s_1}, \cdots, v_{s_n})$	argument
e	::=	v	variable
	|	$(e_1, \cdots, e_n)$	tuple expression
	|	$\lambda v_s . e$	lambda expression
	|	$e_1\ e_2$	application
	|	**let** $v_{s_1} = f_1\ e_1; \cdots; v_{s_n} = f_n\ e_n$ **in** e	let expression
p	::=	v	variable
	|	$C\ (p_1, \cdots, p_n)$	data constructor pattern

Figure 1. The language

3 A Practical Tupling Theorem

The Mutu Tupling Theorem is attractive in the sense that tupling transformation turns out to be a simple symbolic manipulation, as seen in its application to *deepest* in Section 2. It, however, still far from being practical. The most serious problem is that the functions to be tupled must be defined in a specified restrictive way, which cannot be accepted in our functional programming. Nevertheless, the Mutu Tupling Theorem actually guides us to give our main tupling theorem, which can be applied to any lazy functional program for eliminating *multiple data traversals*.

3.1 Programs

To demonstrate our techniques, we use the lazy functional language given in Figure 1. A program is a sequence of function definitions. Functions may be non-recursive or recursive, and recursive functions are defined in a pattern-matching style. This language is nothing special except that we assume that our recursive functions are inductively defined over a *single* (not multiple) recursive parameter and this recursive parameter is (without loss of generality) the first parameter of the function, though we expect our algorithm to have a natural extension to the more general case. To enhence readability, we take liberty to use some familiar syntactic sugars, such as the infix notation, function composition and even case analysis structure as in the definition of *deepest*. In addition, we assume that the renaming has been done whenever there is any danger of name-conflicts.

As an example, consider the following program in our language naively solving the well-known *repsort* problem [Bir84]: transforming a binary tree into one of the same shape in which the leaf values have to be *rep*laced with those of the original tree but arranged in increasing order.

$$repsort\ t = rep\ t\ (sort\ (leaves\ t\ []))$$

where

$$\begin{aligned}
&rep\ (Leaf\ (n))\ ms\ \ = Leaf\ (head\ ms)\\
&rep\ (Node(l,r))\ ms = Node(rep\ l\ (take\ (size\ l)\ ms),\\
&\qquad\qquad\qquad\qquad rep\ r\ (drop\ (size\ l)\ ms))\\
&leaves\ (Leaf\ (n))\ xs\ = n : xs\\
&leaves\ (Node(l,r))\ xs = leaves\ l\ (leaves\ r\ xs)\\
&size\ (Leaf\ (n))\ \ = 1\\
&size\ (Node(l,r)) = plus(size\ l, size\ r)
\end{aligned}$$

In this program the tree is traversed a first time in order to discover and sort the list of leaf values. The tree is then traversed a second time with function *rep*. This function selects appropriate thunks of the sorted sequence in order to pass them on to the left and right subtrees ($take\ k\ x$ takes the first k elements from list x and $drop\ k\ x$ drops the first k elements from list x). At each step, the number of values selected depends on the size of the left subtree, so implicit in the algorithm is a third traversal determining sizes. This is a $O(n^2)$ algorithm in the worst case, n being the number of leaves.

This problem is of interest because there are efficient but non-obvious programs as given in [Bir84, Tak87]. We shall adopt it as our running example and demonstrate how our tupling calculation can provide a systematic way to derive a similar efficient version without multiple data traversals, while only informal studies were given before.

3.2 Main Theorem

To propose our main theorem, we should be more precise about multiple data traversals that are expected to be removed by our tupling calculation.

Definition 1 (Multiple Data Traversal) An expression e is said to have *multiple data traversals* if there exist at least two occurrences of recursive calls like:

$$f\ p \quad \text{and} \quad f'\ p'$$

where p and p' are pattern expressions[4] and p is equal to or a sub-pattern of p'. □

The intuitive observation behind this definition is that the common data p would be traversed twice; once by f and again by f'. Particularly, if f is the same as f', we usually say that the expression e has *redundant recursive calls* to f. For example, the following two expressions

$$\begin{aligned}
&\underline{rep\ \mathrm{t}}\ (sort\ (\underline{leaves\ \mathrm{t}}\ []))\\
&Node(\underline{rep\ \mathrm{l}}\ (take\ (\underline{size\ \mathrm{l}})\ ms), rep\ r\ (drop\ (\underline{size\ \mathrm{l}})\ ms))
\end{aligned}$$

in our running program have multiple data traversals as underlined.

[4] A *pattern expression* is an expression constructed by variables and data constructors, looking like a pattern. That's why we'd like to use p to represent a pattern expression.

We don't guarantee to eliminate multiple data traversals by any functions in a program, because it is impossible to tuple any two arbitrary recursive functions. Therefore some restrictions on recursions should be placed. To this end, we shall define our *tuplable functions* that are potentially tuplable with other functions and enjoy many useful transformation properties (see Section 4) for calculating efficient versions for themselves and for a tuple of themselves.

Definition 2 (Tuplable Function) Assume that $f_1, \cdots, f_m$ are mutually defined by equations:

$$f_i\ p_{ij}\ v_{s_1}\ \cdots\ v_{s_{n_i}} = e_{ij},\ (i = 1, \cdots, m;\ j = 1, \cdots, l_i).$$

The $f_1, \cdots, f_m$ are called *tuplable* (*recursive*) *functions*, if for every occurrence of recursive calls to $f_1, \cdots, f_m$ in all e_{ij}'s, say $f_k\ e'\ e_1\ \cdots\ e_{n_k}$, e' is a sub-pattern of p_{ij}. □

The restriction we impose on the definition of tuplable functions is that the recursive parameter of the tuplable functions should be *strictly decreasing* (i.e., a sub-pattern) across successive recursive calls to themselves. For example, it is easy to check the two equations for the definition of *rep* and see that *rep* is a tuplable function; e.g., for the second equation:

$$\begin{aligned} rep\ (\underline{Node(\mathrm{l},\mathrm{r})})\ ms = Node(&rep\ \underline{\mathrm{l}}\ (take\ (size\ l)\ ms),\\ &rep\ \underline{\mathrm{r}}\ (drop\ (size\ l)\ ms)) \end{aligned}$$

the parameters of all the recursive calls to *rep* in the RHS as underlined are sub-patterns of $Node(l, r)$, the pattern in the LHS. But it excludes the function like *foo* defined by

$$foo(x_1 : x_2 : xs) = x_1 + foo\underline{(2 * x_2 : xs)} + foo\underline{(x_1 : xs)}$$

because two underlined parts are not sub-patterns of $(x_1 : x_2 : xs)$.

Though restricted, tuplable functions cover most of intersting recursive functions in our usual functional programming. As a matter of fact, they cover so-called *mutumorphisms* [Fok89, Fok92] which are considered to be the most general and powerful recursive form over data structures, including catamorphisms and paramorphisms (primitive recursive functions) [Mee92] as their special cases.

Now we are ready to give our main theorem.

Theorem 2 (Main) All multiple data traversals by tuplable functions in a program can be eliminated by tupling calculation. □

In other words, our main theorem says that given a program we can perform tupling calculation to obtain another equivalent version without multiple data traversals *by tuplable functions*. We will prove the theorem later by proposing our tupling calculation algorithm. Returning to our running example, we can calculate the following efficient version for *repsort* in which all multiple traversals over the tree structure by tuplable functions *rep*, *leaves* and *size* are eliminated.

$$\begin{aligned} repsort\ t\ \ =\ \ &\mathbf{let}\ (t_{11}, t_{12}) = g_{\mathcal{G}_1}\ t\\ &\mathbf{in}\ t_{11}\ (sort\ (t_{12}\ [\,]))\end{aligned}$$

where

$$\begin{aligned} g_{\mathcal{G}_1}\ t\ \ &=\ \ \mathbf{let}\ ((r,s),l) = g'_{\mathcal{G}_1}\ t\ \mathbf{in}\ (r,l)\\ g'_{\mathcal{G}_1}\ (Leaf\ (n))\ \ &=\ \ ((\lambda ms.\ Leaf\ (head\ ms), 1), \lambda xs.\ n : xs)\\ g'_{\mathcal{G}_1}\ (Node(l,r))\ \ &=\ \ \mathbf{let}\ ((rl,sl),ll) = g'_{\mathcal{G}_1}\ l\\ &\qquad\qquad ((rr,sr),lr) = g_{\mathcal{G}_1}\ r\\ &\qquad\quad \mathbf{in}\ ((\lambda ms.\ Node(rl\ (take\ sl\ ms),\\ &\qquad\qquad\qquad\qquad rr\ (drop\ sl\ ms)),\\ &\qquad\qquad\quad plus\ (sl, sr)),\\ &\qquad\qquad\quad \lambda xs.\ (ll\ (lr\ xs)))\end{aligned}$$

4 Manipulating Tuplable Functions

Before addressing the proof of the main theorem, we should investigate transformation properties of tuplable functions in order to manipulate them.

4.1 Standardizing Tuplable Functions into Manipulable Form

First of all, we standardize tuplable functions into a manipulable form by making full use of functors in capturing recursive calls to tuplable functions in their definitions.

Lemma 3 (Standardizing) Every tuplable function f can be transformed into the following form:

$$f = \phi \circ (Fh \circ out_F\ \triangle\ F^2h \circ out_F^2\ \triangle\ \cdots\ \triangle\ F^lh \circ out_F^l) \quad (1)$$

where

(i) $h = f_1\ \triangle\ \cdots\ \triangle\ f_n\ \triangle\ g_1\ \triangle\ \cdots\ \triangle\ g_m$, where $f_1, \cdots, f_n$ denote functions mutually defined with f and one of them is f, and $g_1, \cdots, g_m$ denote tuplable functions in the definition of f while traversing over the same recursive data as f;

(ii) $F^n = F^{n-1} \circ F$ and $out_F^n = F^{n-1}out_F \circ out_F^{n-1}$;

(iii) l is a finite natural number.

Proof Sketch: Intuitively, out_F^i can be considered as unfolding i steps of input data, and $F^if\ \circ\ out_F^i$ can be considered as mapping f to all recursive parts of the input data that has been unfolded i steps. Therefore, this lemma reads that one can extract all the recursive calls to the tuplable functions in the definition of f and embed them in the expression

$$Fh \circ out_F\ \triangle\ F^2h \circ out_F^2\ \triangle\ \cdots\ \triangle\ F^lh \circ out_F^l.$$

According to the restriction in the definitions of tuplable functions that the parameters of recursive calls to mutually-defined functions, $f_1, \cdots, f_n$, should be sub-parts of input, it soon follows that every recursive calls to f_i in the RHS can be embedded in a term

$$F^j(f_1\ \triangle\ \cdots\ \triangle\ f_n) \circ out_F^j$$

for $j \in \{1, 2, \cdots, l\}$, where l denotes the maximum number of unfolding steps of the input data for all recursive calls to $f_1, \cdots, f_n$ in the definition. Similarly, the recursive calls to other tuplable functions, say $g_1, \cdots, g_m$, which are on the same data as f can be embedded in an term

$$F^j(g_1\ \triangle\ \cdots\ \triangle\ g_m) \circ out_F^j$$

for $j \in \{1, 2, \cdots, l\}$[5]. To summarize, all recursive calls on the same data as f in the original definition body can be covered by the expression [6]

$$Fh \circ out_F \vartriangle F^2 h \circ out_F^2 \vartriangle \cdots \vartriangle F^l h \circ out_F^l.$$

Thus f can be transformed into the form of (1) by using ϕ to combine recursive calls with other parts forming the body of the original definition. □

As an example, consider how to standardize the definition of tuplable function *rep* into the form of (1). First, we move the non-recursive parameters from the LHS to the RHS by lambda abstraction:

$$\begin{aligned} rep\ (Leaf\,(n)) &= \lambda ms.\, Leaf\,(head\ ms) && (2)\\ rep\ (Node(l,r)) &= \lambda ms.\, Node(rep\ l\ (take\ (size\ l)\ ms), && (3)\\ &\qquad\qquad rep\ r\ (drop\ (size\ l)\ ms)) \end{aligned}$$

From the patterns in the LHS, we see that the input data need at most one step of unfolding, thus we have $l = 1$. The Standardizing Lemma tells us that *rep* can be described in the following form:

$$rep = \phi rep \circ (F_T h \circ out_{F_T}) \qquad (4)$$

Here, h should be a tuple of two parts; all functions mutually defined with *rep* and all other tuplable functions traversing over the same tree as *rep* does, and thus $h = rep \vartriangle size$. Let's see how to calculate ϕrep. Assume $\phi rep = \phi_1 \triangledown \phi_2$.

$$\begin{aligned} rep\ (Leaf\,(n)) &= \{\text{ Equation (4) }\}\\ &\quad (\phi rep \circ (F_T h \circ out_{F_T}))\ (Leaf\,(n))\\ &= \{\text{ Definition of } out_{F_T}\ \}\\ &\quad (\phi rep \circ F_T h)\ (1, n)\\ &= \{\ F_T\ f = id + f \times f\ \}\\ &\quad \phi rep\ (1, n)\\ &= \{\text{ Assumption }\}\\ &\quad \phi_1\ n\\ rep\ (Node\ (l,r)) &= \{\text{ Equation (4) }\}\\ &\quad (\phi rep \circ (F_T h \circ out_{F_T}))\ (Node(l,r))\\ &= \{\text{ Definition of } out_{F_T}\ \}\\ &\quad (\phi rep \circ F_T h)\ (2, (l,r))\\ &= \{\ F_T\ f = id + f \times f\ \}\\ &\quad \phi rep\ (2, (h\ l, h\ r))\\ &= \{\text{ Assumption }\}\\ &\quad \phi_2\ (h\ l, h\ r)\\ &= \{\text{ Definition of } h\ \}\\ &\quad \phi_2\ ((rep \vartriangle size)\ l, (rep \vartriangle size)\ r)\\ &= \{\text{ Expansion }\}\\ &\quad \phi_2((rep\,l, size\,l), (rep\,r, size\,r)) \end{aligned}$$

Comparing the above with Equations (2) and (3) gives:

$$\begin{aligned} \phi_1\ n &= \lambda ms.\, Leaf\,(head\ ms)\\ \phi_2\ ((rl,sl),(rr,sr)) &= \lambda ms.\, Node(rl\ (take\ sl\ ms),\\ &\qquad\qquad rr\ (drop\ sl\ ms)). \end{aligned}$$

[5] Generally it is possible for $j \leq 0$. But this case can be easily removed through simple unfoldings of $g_1, \cdots, g_m$, because $g_1, \cdots, g_m$ are tuplable functions.

[6] If these data are applied by some non-tuplable function, say $k\ e$ where k is a non-tuplable function, we insert the special tuplable function $id = ([in_F])_F$, *identity function*, between them as $k\ (id\ e)$.

In fact, the above standardizing procedure can be made automatic without much difficulty, although the exact algorithm is omitted here. Below are some other examples.

$$\begin{aligned} leaves &= \phi_{leaves} \circ F_T\ leaves \circ out_{F_T}\\ &\quad \textbf{where}\quad \phi_{leaves} = (\lambda n.\lambda xs.\, n : xs) \triangledown\\ &\qquad\qquad (\lambda(l,r).\lambda xs.\, l\,(r\ xs))\\ size &= \phi_{size} \circ F_T\ size \circ out_{F_T}\\ &\quad \textbf{where}\quad \phi_{size} = 1 \triangledown plus \end{aligned}$$

An example that needs unfolding of input data more than one step is the naive definition for *fib* function over the Natural Numbers (see Section 4.4).

4.2 Calculating Tuplable Functions

The main advantage we take from the above standardization process is that tuplable functions become manipulable.

Figure 2 gives some useful transformation rules for manipulating tuplable functions in the form of (1). Rule (R1) shows how to increase l, Rule (R2) shows how to add a new function h_{n+1} to h', and Rule (R3) shows how to exchange positions of functions inside h. Furthermore, Rule (R4) generalizes the Mutu Tupling Theorem.

In the rest of this section, we shall show how to improve tuplable functions through the elimination of multiple data traversals by calculation. Let f be a tuplable function and x be the data over which f traverses. Take a look at a tuplable function f in our standard form:

$$f = \phi \circ (Fh \circ out_F \vartriangle F^2 h \circ out_F^2 \vartriangle \cdots \vartriangle F^l h \circ out_F^l).$$

There are two possibilities that the definition may have multiple traversals over x:

(a) h is a tuple of several tuplable functions some of which are different from f. In this case, these tuplable functions traverse over the same x as f.

(b) $l > 1$. In this case, the computation of $Fh^i \circ out_F^i$ potentially covers that of $F^{i+1}h \circ out_F^{i+1}$ (i.e., redundant recursive calls).

Therefore, to remove multiple data traversals of x in the definition of f requires us to find a suitable way to remove these two possibilities.

To remove the possibility (a), we derive from f a new tuplable function h_{max}. It tuples all tuplable functions that traverse over the same data structures as f, including f as part of its computation, i.e.,

$$f = \Pi \circ h_{max}$$

where Π denotes a projection function[7] . To do so, assuming that $h = f_1' \vartriangle \cdots \vartriangle f_s'$, we reexpress every tuplable function in h into our standard form:

$$f_i' = \phi_i \circ (Fh_i \circ out_F \vartriangle F^2 h_i \circ out_F^2 \vartriangle \cdots \vartriangle F^{l_i} h_i \circ out_F^{l_i})$$

Repeat this procedure for the new h_i until we find

$$h_{max} = f_1' \vartriangle \cdots \vartriangle f_s' \vartriangle \cdots \vartriangle f_r',$$

[7] We define a projection function as an expression built by the projections $\pi_i's$, the identity function id, the function composition $\circ$ and the product $\times$.

$$\frac{f = \phi \circ (Fh \circ out_F \vartriangle \cdots \vartriangle F^l h \circ out_F^l)}{f = (\phi \circ (\pi_1 \vartriangle \cdots \vartriangle \pi_l)) \circ (Fh \circ out_F \vartriangle \cdots \vartriangle F^l h \circ out_F^l \vartriangle F^{l+1} h \circ out_F^{l+1})} \quad \text{(R1)}$$

$$\frac{f = \phi \circ (F(h_1 \vartriangle \cdots \vartriangle h_n) \circ out_F \vartriangle \cdots \vartriangle F^l(h_1 \vartriangle \cdots \vartriangle h_n) \circ out_F^l)}{\begin{array}{rcl} f & = & (\phi \circ (F\Pi \times \cdots \times F^l\Pi)) \circ (FH \circ out_F \vartriangle \cdots \vartriangle F^l H \circ out_F^l) \\ & & \textbf{where } \Pi = \pi_1 \vartriangle \cdots \vartriangle \pi_n,\ H = h_1 \vartriangle \cdots \vartriangle h_n \vartriangle h_{n+1} \end{array}} \quad \text{(R2)}$$

$$\frac{f = \phi \circ (F(h_1 \vartriangle h_2) \circ out_F \vartriangle \cdots \vartriangle F^l(h_1 \vartriangle h_2) \circ out_F^l)}{\begin{array}{rcl} f & = & (\phi \circ (Fex \times \cdots \times F^l ex)) \circ (F(h_2 \vartriangle h_1) \circ out_F \vartriangle \cdots \vartriangle F^l(h_2 \vartriangle h_1) \circ out_F^l) \\ & & \textbf{where } ex\,(x,y) = (y,x) \end{array}} \quad \text{(R3)}$$

$$\frac{\begin{array}{rcl} f & = & \phi \circ (F(f \vartriangle g) \circ out_F \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^l(f \vartriangle g) \circ out_F^l) \\ g & = & \psi \circ (F(f \vartriangle g) \circ out_F \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^l(f \vartriangle g) \circ out_F^l) \end{array}}{f \vartriangle g = (\phi \vartriangle \psi) \circ (F(f \vartriangle g) \circ out_F \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^l(f \vartriangle g) \circ out_F^l)} \quad \text{(R4)}$$

Figure 2. Basic Rules for Manipulating Tuplable Functions

covering all functions in h's part of each $f_i's$ definition.

Now we may assume that $f = \pi_i \circ h_{max}$ for some i. By Rule (R1), (R2) and (R3), we adapt each definition of f_i' to the form of

$$f_i' = \phi_i' \circ (Fh_{max} \circ out_F \vartriangle F^2 h_{max} \circ out_F^2 \vartriangle \cdots \vartriangle F^{l_{max}} h_{max} \circ out_F^{l_{max}}).$$

According to the generalized Mutu Tupling Theorem (i.e., Rule (R4)), we can tuple all of them and obtain:

$$\begin{array}{rcl} h_{max} & = & \phi \circ (Fh_{max} \circ out_F \vartriangle \\ & & \quad F^2 h_{max} \circ out_F^2 \vartriangle \\ & & \quad \cdots \vartriangle \\ & & \quad F^{l_{max}} h_{max} \circ out_F^{l_{max}}) \end{array}$$

where $\phi = \phi_1' \vartriangle \cdots \vartriangle \phi_r'$.

After solving (a), we are able to deal with (b) by introducing $l_{max} - 1$ new functions $u_1, \cdots, u_{l_{max}-1}$:

$$\begin{array}{rcl} u_1 & = & Fh_{max} \circ out_F \\ u_2 & = & Fu_1 \circ out_F \\ & \vdots & \\ u_{l_{max}-1} & = & Fu_{l_{max}-2} \circ out_F \end{array}$$

It follows from the later proof that

$$h_{max} = \phi \circ \eta \circ F(h_{max} \vartriangle u_1 \vartriangle \cdots \vartriangle u_{l_{max}-1}) \circ out_F \quad (5)$$

where $\eta = F\pi_1 \vartriangle \cdots \vartriangle F\pi_{l_{max}}$. By the Mutu Tupling Theorem, we can easily derive that

$$h_{max} \vartriangle u_1 \vartriangle \cdots \vartriangle u_{l_{max}-1} = (\!|\phi \circ \eta \vartriangle F\pi_1 \vartriangle \cdots \vartriangle F\pi_{l_{max}-1}|\!)_F.$$

Therefore, f is transformed to

$$f = (\pi_i \circ \pi_1) \circ (\!|\phi \circ \eta \vartriangle F\pi_1 \vartriangle \cdots \vartriangle F\pi_{l_{max}-1}|\!)_F$$

a composition of a projection function and a catamorphism.

Since both projection functions and catamorphisms contain no multiple data traversals of the input data, we come to a new version of f without multiple data traversals over the input.

Now we return to prove Equation (5) by the following calculation.

$$\begin{array}{ll} & h_{max} = \phi \circ \eta \circ F(h_{max} \vartriangle u_1 \vartriangle \cdots \vartriangle u_{l_{max}-1}) \circ out_F \\ \equiv & \quad \{ \text{ Definition of } \eta \ \} \\ & h_{max} = \phi \circ (F\pi_1 \vartriangle \cdots \vartriangle F\pi_{l_{max}}) \circ \\ & \qquad F(h_{max} \vartriangle u_1 \vartriangle \cdots \vartriangle u_{l_{max}-1}) \circ out_F \\ \equiv & \quad \{ \ \vartriangle \ \} \\ & h_{max} = \phi \circ ((F\pi_1 \circ F(h_{max} \vartriangle u_1 \vartriangle \cdots \vartriangle u_{l_{max}-1})) \vartriangle \\ & \qquad \cdots \vartriangle \\ & \qquad (F\pi_{l_{max}} \circ F(h_{max} \vartriangle u_1 \vartriangle \cdots \vartriangle u_{l_{max}-1})) \\ & \quad) \circ out_F \\ \equiv & \quad \{ \text{ Functor } F \text{ and projection functions } \pi_i\text{'s } \} \\ & h_{max} = \phi \circ (Fh_{max} \vartriangle \\ & \qquad \cdots \vartriangle \\ & \qquad Fu_{l_{max}-1}) \circ out_F \\ \equiv & \quad \{ \ \vartriangle \ \} \\ & h_{max} = \phi \circ (Fh_{max} \circ out_F \vartriangle \cdots \vartriangle Fu_{l_{max}-1} \circ out_F) \\ \equiv & \quad \{ \text{ Expanding the definitions of } u_i\text{'s } \} \\ & \text{True} \end{array}$$

In summary, we have the following theorem.

Theorem 4 (Optimizing Tuplable Function) Let f be a tuplable function and x be the data over which f traverses. f can be calculated into an efficient version in the form of the composition of a projection function and a catamorphism, where multiple traversals over x by tuplable functions are eliminated. □

To make our idea be more concrete, we apply the above algorithm for calculating an efficient version of the tuplable function *rep*. Recall that we have reached the point where

$$\begin{array}{rcl} rep & = & \phi rep \circ F_T(rep \vartriangle size) \circ out_{F_T} \\ size & = & \phi_{size} \circ F_T\ size \circ out_{F_T}. \end{array}$$

We then know that h_{max} is $rep \vartriangle size$ for this case. To find solution to h_{max}, we use Rule (R2) for adapting *size* to the form so that the (generalized) Mutu Tupling Theorem can be applied:

$$size = \phi_{size} \circ F_T \pi_2 \circ F_T\ (rep \vartriangle size) \circ out_{F_T}.$$

It follows from the Mutu Tupling Theorem that

$$rep = \pi_1 \circ (\!|\phi rep \vartriangle (\phi_{size} \circ F_T \pi_2)|\!)_{F_T}.$$

4.3 Tupling Tuplable Functions

Tuplable functions can be optimized by calculation as shown in Theorem 4, resulting in a form of composition of a projection function with a catamorphism. The tuplable functions in this form are suitable to be tupled among each other. That is, tupling these tuplable functions will give a tuplable function in this form again, as stated in the following theorem.

Theorem 5 (Tupling Tuplable Functions) Let

$$f_i = \Pi_i \circ ([\![\phi_i]\!])_F,\ i = 1, \cdots, n$$

be n tuplable functions where Π_i stands for a projection function. Then,

$$f_1 \vartriangle \cdots \vartriangle f_n = \Pi \circ ([\![\phi]\!])_F$$

where $\Pi = \Pi_1 \times \cdots \times \Pi_n$ and $\phi = \phi_1 \circ F\pi_1 \vartriangle \cdots \vartriangle \phi_n \circ F\pi_n$. □

4.4 Another Example

In this section, we consider a classical example often used to illustrate the super-linear speedup achieved by the traditional tupling with the invention of a tuple of two functions. We shall demonstrate that such speedup can be obtained mechanically by our calculation over tuplable functions.

A naive definition of the fibonacci function is:

$$\begin{array}{lcl} \mathit{fib}\ \mathit{Zero} & = & \mathit{Zero} \\ \mathit{fib}\ (\mathit{Succ}(\mathit{Zero})) & = & \mathit{Succ}(\mathit{Zero}) \\ \mathit{fib}\ (\mathit{Succ}(\mathit{Succ}(n))) & = & \mathit{plus}(\mathit{fib}(\mathit{Succ}(n)), \mathit{fib}(n)) \end{array}$$

where fib is a recursion over the natural number data type:

$$N = \mathit{Zero} \mid \mathit{Succ}(N)$$

which is defined by the functor F_N:

$$F_N = !\mathbf{1} + I.$$

Obviously, this definition gives an inefficient exponential algorithm because of many redundant recursive calls to *fib*.

Checking that *fib* is a tuplable function, we soon know that all redundant recursive calls to *fib* can be removed. To do so, as the first step, we standardize the definition. The LHS of the definition of *fib* tells us that we need at most two steps of unfolding of its input. Therefore, according to Lemma 3, *fib* should be standardized to the following.

$$\mathit{fib} = \phi \circ (F_N \mathit{fib} \circ \mathit{out}_{F_N} \vartriangle F_N^2 \mathit{fib} \circ \mathit{out}^2_{F_N})$$

Noting that

$$\begin{array}{lcl} F_N & = & !\mathbf{1} + I \\ F_N^2 & = & !\mathbf{1} + (!\mathbf{1} + I) \\ \mathit{out}_{F_N} & = & \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \to (1,()) \\ & & \qquad\qquad \mathit{Succ}(n) \to (2, n) \\ \mathit{out}^2_{F_N} & = & \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \to (1,()) \\ & & \qquad\qquad \mathit{Succ}(n) \to \\ & & \quad (2, \mathbf{case}\ n\ \mathbf{of}\ \mathit{Zero} \to (1,()) \\ & & \qquad\qquad\qquad \mathit{Succ}(n') \to (2, n')) \end{array}$$

we replace them and get

$$\begin{array}{l} \mathit{fib} = \phi \circ (\lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \to ((1,()),(1,())) \\ \qquad\qquad\qquad\qquad\qquad \mathit{Succ}(n) \to \\ \qquad\qquad\qquad\qquad ((2, \mathit{fib}\, n), \\ \qquad\qquad\qquad\qquad\ (2, \mathbf{case}\ n\ \mathbf{of}\ \mathit{Zero} \to (1,()) \\ \qquad\qquad\qquad\qquad\qquad\ \mathit{Succ}(n') \to (2, \mathit{fib}\, n'))) \end{array}$$

On the other hand, note that the original definition of *fib* can be transformed into the following using the case structure:

$$\begin{array}{lcl} \mathit{fib} & = & \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \to \mathit{Zero} \\ & & \qquad\qquad\qquad\quad \mathit{Succ}(n) \to \\ & & \qquad\quad \mathbf{case}\ n\ \mathbf{of}\ \mathit{Zero} \to \mathit{Succ}(\mathit{Zero}) \\ & & \qquad\qquad\qquad\quad \mathit{Succ}(n') \to \mathit{plus}(\mathit{fib}\, n, \mathit{fib}\, n'). \end{array}$$

Matching the above two definitions of *fib* gives the definition for ϕ:

$$\begin{array}{lcl} \phi & = & \lambda(x,y).\ \mathbf{case}\ (x,y)\ \mathbf{of}\ ((1,()),(1,())) \to \mathit{Zero} \\ & & \qquad\qquad\qquad\qquad\quad ((2,f_1),(2,y')) \to \\ & & \qquad\quad \mathbf{case}\ y'\ \mathbf{of}\ (1,()) \to \mathit{Succ}(\mathit{Zero}) \\ & & \qquad\qquad\qquad\quad (2,f_2) \to \mathit{plus}(f_1, f_2). \end{array}$$

Now according to Theorem 4, we get the result of

$$\mathit{fib} = \pi_1 \circ ([\![\phi \circ (F_N \pi_1 \vartriangle F_N \pi_2) \vartriangle F_N \pi_1]\!])_{F_N}.$$

A little simplification and inlining of the catamorphism will lead to the following efficient linear program:

$$\begin{array}{lcl} \mathit{fib}\ n & = & x \\ & & \mathbf{where}\ (x,y) = f'\ n \\ f'\ \mathit{Zero} & = & (\mathit{Zero}, (1,())) \\ f'\ (\mathit{Succ}(n)) & = & (\mathbf{case}\ y'\ \mathbf{of}\ (1,()) \to \mathit{Succ}(\mathit{Zero}) \\ & & \qquad\qquad\qquad\quad (2,f_2) \to \mathit{plus}(f_1, f_2), \\ & & \quad (2, f_1)) \\ & & \mathbf{where}\ (f_1, y') = f'\ n \end{array}$$

in which all redundant recursive calls to *fib* due to multiple traversals of the input have been successfully eliminated.

5 Tupling Calculation Algorithm

We are now ready to prove our main theorem given in Section 3.2. We shall propose a tupling calculation algorithm which can eliminate all multiple data traversals by tuplable functions by means of calculation.

5.1 The Algorithm

Our tupling calculation algorithm has been summarized in Figure 5. It starts with the function to be optimized, aiming to attain a new version from it such that there are no multiple data traversals by any two tuplable functions.

If the function to be optimized is a tuplable function, this is easy; just applying Theorem 4 to its definition to turn it into a composition of a projection function and a catamorphism. Again we should remember to optimize the function inside the catamorphism.

Otherwise, the function to be optimized is a non-recursive function, i.e.,

$$f\ p\ v_{s_1}\ \cdots\ v_{s_{n'}} = e$$

Tupling Calculation Algorithm $\mathcal{T}$:

1. Start with the function to be optimized.
2. If the function to be optimized has been done, then return.
3. *If the function to be optimized is a tuplable function*, according to Theorem 4 we can perform tupling calculation to turn it into the form of $\Pi \circ (\![\phi]\!)$ where Π denotes a projection function. Then, apply $\mathcal{T}$ for optimizing function ϕ.
4. *If the function to be optimized is a non-tuplable function*, for each equation

$$f\ p\ v_{s_1}\ \cdots\ v_{s_{n'}} = e$$

select out all calls to recursive functions in the form of $f'\ e'$ in e and classify them into two sets: $\mathcal{C}_1$ for the calls to non-tuplable functions; $\mathcal{C}_2$ for the calls to tuplable functions.

- For each function in $\mathcal{C}_1$, if it has not been optimized, apply $\mathcal{T}$ for its optimization.
- For each function in $\mathcal{C}_2$, if it has not been optimized, apply $\mathcal{T}$ for its optimization. We then group recursive functions in $\mathcal{C}_2$; in each group functions are applied to the same expression. Let $\mathcal{G}_1, \cdots, \mathcal{G}_r$ be groups we have got. For every group $\mathcal{G}_i = \{g_{i1}\ e_i, \cdots, g_{in_i}\ e_i\}$, we can calculate an efficient version for $g_{\mathcal{G}_i} = g_{i1} \vartriangle \cdots \vartriangle g_{in_i}$ according to Theorem 5. Let $t_{i1}, \cdots, t_{in_i}$ be fresh variables for each group, we turn the original equation into:

$$\begin{array}{lcll} f\ p\ v_{s_1}\ \cdots\ v_{s_{n'}} & = & \textbf{let} & (t_{11}, \cdots, t_{1n_1}) = g_{\mathcal{G}_1}\ e_1 \\ & & & \cdots \\ & & & (t_{r1}, \cdots, t_{rn_r}) = g_{\mathcal{G}_r}\ e_r \\ & & \textbf{in} & e[g_{ij}\ e_i \mapsto t_{ij},\ \text{for } i = 1, \cdots, r \text{ and } j = 1, \cdots, n_i] \end{array}$$

Figure 3. The Tupling Calculational Algorithm

where e contains no occurrences of f, or is a recursive but not a tuplable function, i.e.,

$$f\ p_i\ v_{s_1}\ \cdots\ v_{s_{n'}} = e_i,\ (i = 1, \cdots, n).$$

We then step to eliminate multiple data traversals by tuplable functions in e (or e_i). This is achieved by optimizing all non-tuplable functions used in e (i.e., the functions in the set $\mathcal{C}_1$) as well as all tuplable functions (i.e., the functions in the set $\mathcal{C}_2$), and then grouping optimized tuplable functions in $\mathcal{C}_2$. Several remarks should be made on this grouping process.

First, for the sake of simplicity we have assumed that all expressions being applied by the tuplable functions in $\mathcal{C}_2$ are not overlapped. In other words, for any two calls $f_1\ e_1$ and $f_2\ e_2$ in $\mathcal{C}_2$, e_1 is not a subpattern of e_2 and vice versa. For example, we do not allow the two calls of

$$f_1\ xs,\ f_2\ (x : xs)$$

because xs is a subpattern of $x : xs$. But this restriction can be removed. Recalling that any tuplable function f can be turned into $\Pi \circ f'$ where $f' = (\![\phi]\!)_F$ according to Theorem 4, for a call like $f\ (in_F\ (e_1, \cdots, e_n))$, we can promote the tuplable function f into the expression $in_F\ (e_1, \cdots, e_n)$, i.e.,

$$f\ (in_F\ (e_1, \cdots, e_n)) = \Pi\ (\phi\ (F\ f'\ (e_1, \cdots, e_n)))$$

distributing f' to some e_i's by F.

Second, thanks to the above assumption, we are able to restrict ourselves to the tupling of the functions that are applied to the *same* expressions in $\mathcal{C}_2$, as shown in the algorithm.

To see how the algorithm works practically, consider our running example *repsort*. We start with optimizing *repsort*. As it is not a recursive definition, we go to Step 2. For the equation

$$repsort\ t = rep\ t\ (sort\ (leaves\ t))$$

we select out all calls to recursive functions in RHS and get two recursive calls, namely $rep\ t$ and $leaves\ t$. As they are tuplable functions, we then have $\mathcal{C}_1 = \{\}$ and $\mathcal{C}_2 = \{rep\ t, leaves\ t\}$.

Now we process on $\mathcal{C}_2$. First, we should optimize all tuplable functions, *rep* and *leaves*, that appear in $\mathcal{C}_2$. To optimize *rep*, we go to Step 3. As shown in Section 4.2, we have got the following solution for *rep* (after a straightforward simplification):

$$\begin{array}{lcl} rep & = & \pi_1 \circ (\![\phi_1' \triangledown \phi_2']\!) \\ \phi_1'\ n & = & (\lambda ms.\ Leaf\ (head\ ms), 1) \\ \phi_2'\ ((rl, sl), (rr, sr)) & = & (\lambda ms.\ Node(rl\ (take\ sl\ ms), \\ & & \qquad rr\ (drop\ sl\ ms)), \\ & & \quad plus\ (sl, sr)) \end{array}$$

Here we need to optimize ϕ_1' and ϕ_2' in order to get the final efficient version for *rep*. We don't address this optimization here, which is similar to *repsort*. In fact, if *take* and *drop* are defined as tuplable functions, we can tuple them when optimizing ϕ_2'. So much for *rep*. Similarly, we can optimize *leaves* and have the result given in Section 4.2. Next, we should group recursive calls in $\mathcal{C}_2$. Here we only have a single group $\mathcal{G}_1 = \{rep\ t, leaves\ t\}$ in which all tuplable functions are applied to the same t. So we define $g_{\mathcal{G}_1} = rep \vartriangle leaves$.

Table 1. Experimental Results (*repsort*)

Input Size (leaves no.)	**Time** (secs/10 times) before tupling	after tupling	**Allocations** (bytes) before tupling	after tupling
800	0.08	0.06	95,920	157,544
1,600	0.18	0.08	198,100	313,740
3,200	0.38	0.18	410,068	626,060
6,400	0.84	0.30	849,364	1,250,700
12,800	1.76	0.58	1,758,916	2,500,136

According to Theorem 5, we can calculate that

$$g_{\mathcal{G}_1} = (\pi_1 \times id) \circ (\!|((\phi_1' \triangledown \phi_2') \circ F_T \pi_1) \vartriangle (\phi_{leaves} \circ F_T \pi_2)|\!)_{F_T}$$

By introducing two new variables t_{11} and t_{12}, we thus have obtained our result:

$$\begin{array}{lcl} repsort\, t & = & \textbf{let } (t_{11}, t_{12}) = g_{\mathcal{G}_1}\ t \\ & & \textbf{in } t_{11}\ (sort\ (t_{12}\ [\,])). \end{array}$$

Summarizing all above, inlining the definition of $g_{\mathcal{G}_1}$ and a little simplification will lead to the familiar program given in Section 3.2, where all multiple traversals over trees have been successfully eliminated, which is as efficient as those in [Bir84, Tak87].

5.2 Properties of the Algorithm

Our tupling calculational algorithm enjoys many important properties. First, our algorithm is correct. This follows from the fact that each step of our transformation is based on meaning-preserved transformation rules and theorems.

Second, our algorithm terminates. This is because (i) the number of functions to be optimized by our algorithm is limited for a given program. Basically, our algorithm optimizes those functions which are used directly or indirectly by the main function, and (ii) each application of transformation rules in the algorithm terminates; e.g., the two significant transformations based on Theorem 4 and 5 terminate as easily verified.

Third, our algorithm guarantees that all multiple data traversals *by tuplable function* can be eliminated. It expects spectacular efficiency improvement under the lazy evaluation, in the sense that the new program obtained from the algorithm is faster than the original. But this speedup comes at the price of using some extra memory for tupling and with an assumption of existence of an efficient implementation of tuple.

The most significant speedup attained from our tupling calculation algorithm is due to the removal of all redundant recursive calls to tuplable functions in a program. Take a look at the examples in this paper. Our algorithm:

- eliminates the redundant recursive calls to *depth* in *deepest*, giving a linear algorithm from the original quadratic one;
- eliminates the redundant recursive calls to *fib*, giving a linear algorithm from the original exponential one;
- eliminates the redundant recursive calls to *size* in *rep*, giving a linear *rep* from the original quadratic one.

To be more concrete, we evaluate two versions of *repsort* before and after tupling transformation in this paper using the Glasgow Haskell compiler (version ghc-0.29). Table 1 shows the experimental results. The input to the programs for our testing are balanced binary trees, thus the number of their leaves can be used as a measure of input size. The execution time and the allocations are obtained using the profiling mechanism provided by the Glasgow Haskell Compiler. It can be seen that with the size of the input tree going larger the new version obtained by our tupling transformation becomes much more faster than the original one, but with about 50% additional allocations.

It is worth noting that by the ordinary implementation of the tuple data structure [Pey88] our algorithm do not guarantee spectacular efficiency improvements. This is a common problem among all existing tupling methods. Consider for example the following *average* function:

$$\begin{array}{lcl} average\ x & = & sum\, x\ /\ length\, x \\ & & \textbf{where} \\ & & \quad sum = foldr\ (+)\ 0 \\ & & \quad length = foldr\ (\lambda x \lambda y.1 + y)\ 0 \end{array}$$

which can be transformed into the following by our algorithm:

$$\begin{array}{l} average'\ x = \textbf{let } (s,l) = sl\ x \textbf{ in } s/l \\ \quad \textbf{where } sl = foldr\ (\lambda x \lambda \underline{(s,l)}.\underline{(x+s, 1+l)})\ (0,0). \end{array}$$

Obviously, *average'* wins over *average* in that it reduces twice traversals of the input list x by *sum* and *length* respectively into once. On the other hand it also increases time cost (besides space cost) for unpacking and packing a pair when sl traverses over x. If the cost is bigger than what we gain by tupling transformation, the transformed program will be slower than the original. The *average* indeed gives such an example.

We will not be involved in solving this problem in this paper. It would be interesting to see that *average* should lead to efficiency improvement (in time) under the assumption that the program

$$(\!|\phi \circ F\pi_1 \vartriangle \psi \circ F\pi_2|\!)$$

can be more efficiently implemented than

$$(\!|\phi|\!)_F \vartriangle (\!|\psi|\!)_F.$$

This assumption requires an efficient implementation of tuple, avoiding the cost for packing and unpacking of the tupled value.

6 Related Work and Discussion

The use of generic control structures which capture patterns of recursions in a uniform way is of great significance in program transformation and optimization [MFP91, Fok92, SF93, TM95]. Our work is much related to these studies. In particular, our work was greatly inspired by the success of applying this approach to fusion transformation as studied in [SF93, GLJ93, LS95, TM95].

We made the first attempt to apply this calculational approach to the tupling transformation. Previous work, as intensively studied by Chin [Chi93], tries to tuple *arbitrary* functions by *fold/unfold* transformations. In spite of its generality, it has to keep track of all the function calls and devise clever control to avoid infinite unfolding, resulting in high runtime cost which prevents it from being employed in a real compiler system. We follow the experience of work of fusion in calculational forms [TM95, HIT96b] and base our tupling theorem on a simple calculational rule: the Mutu Tupling Theorem. We define a class of tuplable functions which is more general than Chin's **T0** Class[Chi93] in that we allow the parameters other than the recursive one to be arbitrary. Chin needs such restriction in order to guarantee the termination of fold/unfold transformation, while termination is not a problem in our approach. In [Chi93], Chin focused on the tupling analysis to find what functions should be tupled but he didn't show how the efficient program can be obtained in a more systematic way. In sharp contrast, we propose a concrete tupling algorithm to calculate efficient version eliminating multiple data traversals. Though being simple and less general than Chin's, our tupling transformation, as demonstrated, can be applied to improve a wide class of functions. More important, our algorithm should be easily implemented, which promises to be used in a practical compiler.

Tupling and fusion are two much related transformations for improving functional programs. It is worth noting that our tupling algorithm can well coexist with fusion under the transformation in calculational form. Our tupling algorithm improves the recursion by constructing a catamorphism (Theorem 4), making ease for fusion transformation. A relevant study can be found in [HIT96a] where tupling and fusion are used together to derive list homomorphisms (i.e., catamorphisms over append lists).

Elimination of multiple traversals over data structures has been studied for a long time. Our work is related to these works. Bird [Bir84] suggested the use of circular programs. Takeichi [Tak87] used a different technique called *lambda hoisting* with introduction of common higher order functions. In particular, we are much influenced by Pettorossi's work [Pet87] of using lambda abstraction in conjunction with the tupling tactic. However, the transformations in the previous work require more or less human-insights, which are hard to be made automatic. Other related work includes memoization [Mic68, Hug85], tabulation [Bir80, Coh83, CH95] and incremental algorithms [Liu96].

All transformation algorithms introduced in this paper have been implemented in a rapid prototyped way. It is completely mechanical and does not rely on heuristics. Although we have to wait for the detailed experimental results to say that this system is effective for practical programs, we are absolutely convinced that our calculational approach to tupling transformation makes a good progress in code optimization of functional programs. In addition, our tupling calculational transformation is expected to be added to the HYLO system [OHIT97], a calculational system for improving functional programs, which is now under development in the University of Tokyo.

Acknowledgement

This paper owes much to the thoughtful and helpful discussions with Fer-Jan de Vries, Masami Hagiya, Muzuhito Ogawa and other members of the Tokyo CACA seminars. Thanks are also to referees who provided detailed and helpful comments.

References

[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. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

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

[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.

[Coh83] N.H. Cohen. Eliminating redundant recursive calls. *ACM Transaction on Programming Languages and Systems*, 5(3):265–299, July 1983.

[Fok89] M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.

[Fok92] M. Fokkinga. *Law and Order in Algorithmics.* Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[Hag87] T. Hagino. *Category Theoretic Approach to Data Types.* Ph.D thesis, University of Edinburgh, 1987.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms via tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418, Cracow, September 1996. Springer-Verlag.

[HIT96b] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA, May 1996. ACM Press.

[Hug85] J. Hughes. Lazy memo-functions. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 201), pages 129–149, Nancy, France, September 1985. Springer-Verlag, Berlin.

[Liu96] Y.A. Liu. *Incremental Computation: A Semantics-Based Systematic Transformation Approach*. PhD thesis, Department of Computer Science, Cornell University, 1996.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, California, June 1995.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, August 1991.

[MH95] E. Meijer and G. Hutton. Bananas in space: Exteding fold and unfold to exponential types. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, California, June 1995.

[Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[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*, 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).

[Pey88] S.L. Peyton Jones. *The implementation of functional programming languages*. Prentice-Hall, 1988.

[PP91] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic progrrams. In *3rd Int'l Symp., PLILP'91*, pages 247–258, Passau, Germany, August 1991. LNCS 528.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[Tak87] M. Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.