Lightweight Fusion by Fixed Point Promotion *

(the author's version)

Atsushi Ohori Isao Sasano †

Research Institute of Electrical Communication
Tohoku University
{ohori, sasano}@riec.tohoku.ac.jp

Abstract

This paper proposes a lightweight fusion method for general recursive function definitions. Compared with existing proposals, our method has several significant practical features: it works for general recursive functions on general algebraic data types; it does not produce extra runtime overhead (except for possible code size increase due to the success of fusion); and it is readily incorporated in standard inlining optimization. This is achieved by extending the ordinary inlining process with a new fusion law that transforms a term of the form $f \circ (\texttt{fix}\, g.\lambda x.E)$ to a new fixed point term $\texttt{fix}\, h.\lambda x.E'$ by promoting the function f through the fixed point operator. This is a sound syntactic transformation rule that is not sensitive to the types of f and g. This property makes our method applicable to wide range of functions including those with multi-parameters in both curried and uncurried forms. Although this method does not guarantee any form of completeness, it fuses typical examples discussed in the literature and others that involve accumulating parameters, either in the `foldl`-like specific forms or in general recursive forms, without any additional machinery. In order to substantiate our claim, we have implemented our method in a compiler. Although it is preliminary, it demonstrates practical feasibility of this method.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization; D.3.4 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Program and recursion schemes

General Terms Languages, Theory

Keywords Fusion, inlining, fixed point

* This research was partially supported by the Japan MEXT (Ministry of Education, Culture, Sports, Science and Technologies) leading project of "Comprehensive Development of Foundation Software for E-Society" under the title "dependable software development technology based on static program analysis."

† The second author was also partially supported by Grant-in-Aid for Young Scientists (B), 16700029 from the Ministry of Education, Culture, Sports, Science and Technology.

1. Introduction

Fusion is a program optimization technique to combine two adjacent computations, where one produces a result which is then processed by the other, by "fusing" two successive computation steps (function applications or loops) into one, yielding a more efficient program. In functional programming, this technique has the additional significance that it suppresses construction of intermediate data structures, such as lists and other inductively defined data structures. Burstall and Darlington [1] were perhaps the first to study this optimization in functional programming and presented a general strategy for generating efficient recursive programs through fold/unfold transformation. Wadler [23] observed that this optimization can be carried out systematically to eliminate intermediate trees altogether and showed the "deforestation" theorem stating that every composition of functions in a certain restricted form can be effectively fused into a single function. Since this seminal observation, a series of investigations have been done towards establishing a general method for program fusion.

One direction toward developing a practical fusion method is to restrict the target functions to be of specific forms. Gill, Launchbury and Peyton Jones [6] presented the "short cut" deforestation law stating that if both the producer function and the consumer function are written in a specific form then they are fused together. Since this rule is a simple local transformation, it is easily incorporated in an optimizing compiler. This approach has been generalized to user-defined recursive data types [19, 22].

One major limitation of these approaches is that they do not deal with general recursive functions. While it is certainly a valid claim that "lucid" or compositional style programming has the advantages of clarity and modularity, one often has to define data structures specific to the problem domain and to write specialized functions using general recursion. In serious practical software development, this tendency is predominant. Even in the GHC compiler for Haskell, which implements the rule for short cut fusion [17], build/cata style code appears to be minority compared with general recursive functions. Since the general idea underlying deforestation is equally applicable to general recursive definitions, it is highly desirable to develop a fusion method that works directly on general recursive functions.

Launchbury and Sheard [12] proposed one solution to this problem by developing an algorithm to transform a recursive function definition into a program in build/cata form. The resulting program is then fused by build/cata laws. Chitil [4] proposed a type based approach to obtaining build forms from general recursive definitions. These approaches are conceptually elegant, but their practical feasibility is not clear. A key step in their development is to represent a term that constructs a data structure as a higher-order function using the technique of representing term algebra (induc-

tive types) in the second-order lambda calculus. For example, an efficient and compact list term `1:2:nil` (written in Haskell syntax) is temporarily "heated up" to a lambda term $\lambda c.\lambda n.c\ 1\ (c\ 2\ n)$. It is questionable whether or not these costly abstractions should all be fused away in an optimizing compiler for a language that involves various features over pure lambda expressions.

Another limitation of the short-cut fusion approach is that it is restricted to functions of single argument. In practical software development, functions are often written in efficient tail recursive form using accumulating parameters, or they may take various extra parameters required by the problem domain. Unfortunately, short-cut fusion [6] and its extension [22, 21] do not scale up to those functions.

In order for a fusion method to become a practical optimization method for an optimizing compiler, it should ideally have the following features.

1 It is simple and fully automatic so that it can be easily embedded in an optimizing compiler.

2 It does not introduce any extra runtime overhead.

3 It works for general recursive functions with multiple parameters (including accumulating parameters and others).

4 It works for general user-defined data types.

5 *It fuses all fusable functions.*

Development of such a method is the ultimate goal of the subject. However, if we weaken the requirement by replacing the last condition of completeness to

5' *it works for expected typical cases,*

then we can indeed develop such a system. An attempt is made in this paper to develop such a fusion method.

Previous researches have mainly focused on algebraic (or categorical) properties of generic functions on data structures for deducing general fusion patterns. Some notable examples include the promotion theorem for `foldr` [13] and its generalizations [14, 19], short-cut deforestation for `foldr` and `build` [6] and its generalization in calculational form [22], `destroy` and `unfold` fusion [21], and more recent algebraic fusion based on monoid homomorphism [11]. Instead of pursuing this direction, we follow the original approach of fold/unfold transformation [1] and develop a method to fuse two recursive function definitions directly through unfolding, simplification (beta-reduction), and folding (generating new recursive definitions). The crucial step in developing a practical fusion method is to find deterministic and simple, yet powerful rules to control the fusion process.

By analyzing various fusable recursive functions, we have discovered the following simple yet effective transformation strategy. Let $f = \texttt{fix}\, f.\lambda x.E_f$ and $g = \texttt{fix}\, g.\lambda x.E_g$ be recursive function definitions. Under these bindings, we can transform the composition $f \circ g$ to obtain a fusion of the two as follows.

1. Inline the body of g to obtain $f \circ \lambda x.E_g$, which is beta reduced to $\lambda x.f\ E_g$.
2. Transform $f\ E_g$ to E'_g by distributing the function symbol f to all the tail positions of E_g.
3. Inline f once in E'_g and simplify the term to obtain a term $E_{f,g}$.
4. Replace the occurrences of $f \circ g$ in $E_{f,g}$ by a new function name f_g and generate a new binding $f\_g = \texttt{fix}\, f\_g.\lambda x.E_{f\_g}$.

Since this process promotes the function f through the fixed point operator, we call it *fixed point promotion*.

Step 2 is represented by a term transformation axiom, which we call (AppDist). The entire step is represented by a transformation rule that produces a new fixed point term (Step 4) by reducing the body using (AppDist) and other reduction axioms. We refer to this transformation rule as (FixPromote).

This remarkably simple process can fuse various general recursive functions, including those involving accumulating parameters or tail recursive functions such as `foldl` without resorting to any heuristics. Since this process only involves two simple deterministic transformation rules, it can easily be incorporated in any inlining process. Moreover, it is robust enough to scale up to practical optimizing compiler of a polymorphic functional language. We therefore claim that the resulting fusion algorithm is a practical one that satisfies the desired criteria mentioned above. In order to substantiate this claim, we have implemented the algorithm in a prototype compiler for a full scale functional language, and have tested various examples. The results verify that our method fuses typical examples including those with extra parameters. The implementation is made available through the Internet for downloading. See Section 6 for the details. We should note that the implementation is a preliminary one only for testing the behavior of the proposed method, and does not intend to be a part of a practical optimization phase.

The rest of the paper is organized as follows. Section 2 informally presents our lightweight fusion method through examples. Section 3 defines a simple functional language, gives our two new fusion rules, and proves the soundness of the two rules. Section 4 presents the fusion algorithm. Section 5 demonstrates the power of the algorithm through various examples. Section 6 descries our experimental implementation. Section 7 shows some general benchmark results and their analysis. Section 8 discusses related work. Section 9 concludes the paper.

2. Overview of Lightweight Fusion

To illustrates our method, let us first consider the following simple example.

```
let mapsq =
      fix mapsq.
          λL.case L of
                nil => nil,
                cons(h,t) => cons(h*h, mapsq t)
    sum = fix sum.λL.case L of
                        nil => 0,
                        cons(h,t) => h + sum t
in ··· sum (mapsq E) ··· end
```

On detecting an application `sum (mapsq` E`)` of a composition of two recursive functions, we attempt to generate a new recursive definition starting from the composition function:

```
λx.sum (mapsq x)
```

where the subterms that are transformed in the subsequent step are underlined.

We first inline the lambda body of the inner function `mapsq`.

```
λx.sum (case x of
          nil => nil,
          cons(h,t) => cons (h*h, mapsq t))
```

Inlining is not recursive, and hence `mapsq` in the body refers to the original definition.

Next, we apply the (AppDist) rule: we distribute the function name `sum` to all the *tail positions* of its argument term.

```
λx.case x of
     nil => sum nil,
     cons(h,t) => sum (cons (h*h, mapsq t))
```

We then inline the lambda body of sum once, and simplify the resulting term.

```
λx.case x of
     nil => 0,
     cons(h,t) => h*h + sum (mapsq t)
```

If the resulting term contains the composition `sum (mapsq _)` that we are fusing, then we create a new recursive function name `sum_mapsq` and substitute it for the composition.

```
fix sum_mapsq.λx.case x of
                  nil => 0,
                  cons(h,t) => h*h + sum_mapsq t
```

This entire process is an application of (FixPromote).

The successful application of (FixPromote) indicates that the composition of the two functions is fused into a new recursive function. If this happens, then we put the new definition after the definitions of `sum` and `mapsq`, and replace all the occurrences of the composition `sum` ∘ `mapsq` appearing in the rest of the program by `sum_mapsq`. If there is no reference to the original functions `sum` and `mapsq` then their definitions are removed by dead code elimination, and the following program is obtained.

```
let sum_mapsq =
      fix sum_mapsq.
          λx.case x of
               nil => 0,
               cons(h,t) => h*h + sum_mapsq t
in ··· sum_mapsq E ··· end
```

This process is easily extended to functions of multiple parameters. For example, suppose `sum` is defined in tail recursive form as follows.

```
let mapsq = ...
    sum = fix sum.λL.λS.
                case L of
                  nil => S,
                  cons(h,t) => sum t (S + h)
in ··· sum (mapsq E) 0 ··· end
```

We start with `λx.λS.sum (mapsq x) S` and first inline `mapsq`.

```
λx.λS.sum (case x of
             nil => nil,
             cons(h,t) => cons (h*h, mapsq t)) S
```

We then distribute the application context `sum _ S` to all the tail positions.

```
λx.λS.case x of
        nil => sum nil S,
        cons(h,t) => sum (cons (h*h, mapsq t)) S
```

Next, we inline `sum` once and simplify.

```
λx.λS.case x of
        nil => S,
        cons(h,t) => sum (mapsq t) (S + h*h)
```

Finally, we replace `λx.λS.sum (mapsq x) S` with a new function name `sum_mapsq` and obtain the following program (after dead code elimination).

```
let sum_mapsq =
      fix sum_mapsq.λx.λS.
          case x of
            nil => S,
            cons(h,t) => sum_mapsq t (S + h*h)
in ··· sum_mapsq E 0 ··· end
```

This transformation process also extends to uncurried multi-argument functions.

We note that the process outlined above is simple, terminating and entirely automatic, so that it can easily be embedded in the standard inlining process. Moreover, it relies neither on any heuristics nor on any special properties of data types or functions. It is therefore applicable to general function definitions manipulating user-defined data types.

3. Fixed Point Promotion Laws and its Soundness

This section presents the new fusion laws on which our method is based, and shows their soundness.

3.1 The source language and some notations

We consider the following set of lambda terms.

$$\begin{array}{rcl} M & ::= & x \mid \lambda x.M \mid \texttt{fix}\, f.M \mid M\ M \\ & \mid & C(M,\ldots,M) \\ & \mid & \texttt{case}\ M\ \texttt{of}\ p\texttt{=>}M,\ldots,p\texttt{=>}M \\ & \mid & \texttt{let}\ x = M\ \texttt{in}\ M\ \texttt{end} \\ p & ::= & C(x,\ldots,x) \end{array}$$

$\texttt{fix}\, f.M$ denotes the fixed point of the functional $\lambda f.M$ and represents a recursive function term. $C(M,\ldots,M)$ is a data constructor term for algebraic (user defined) data types. We sometimes write $\texttt{let}\ x_1 = M_1,\ldots,x_n = M_n\texttt{in}\ M\ \texttt{end}$ for an abbreviation of a nested let expression. If M and N are terms, we write $M\{N/x\}$ for the term obtained by the usual capture free substitution of N for x in M. In addition to this core syntax, in examples, we use terms containing primitive operations.

For this language, we assume the usual bound variable convention, i.e., the set of bound variables are pairwise distinct and are different from free variables.

A *context*, ranged over by $\mathcal{C}$, is defined by the following syntax

$$\begin{array}{rcl} \mathcal{C} & ::= & [\,] \mid x \mid \lambda x.\mathcal{C} \mid \texttt{fix}\, f.\mathcal{C} \mid \mathcal{C}\ \mathcal{C} \\ & \mid & C(\mathcal{C},\ldots,\mathcal{C}) \\ & \mid & \texttt{case}\ \mathcal{C}\ \texttt{of}\ p\texttt{=>}\mathcal{C},\ldots,p\texttt{=>}\mathcal{C} \\ & \mid & \texttt{let}\ x = \mathcal{C}\ \texttt{in}\ \mathcal{C}\ \texttt{end} \end{array}$$

where $[\,]$ is a "hole". If we need to identify each hole differently, we use some index i and write $[\,]_i$. A context $\mathcal{C}$ generated by this grammar contains zero or more holes. In what follows, we let $\mathcal{C}$ range only over those contexts that contain *one or more holes*. We write $\mathcal{C}[M_i]_i$ for the term obtained from $\mathcal{C}$ by filling each hole $[\,]_i$ of $\mathcal{C}$ with (different) M_i, and write $\mathcal{C}[M]$ for the term obtained from $\mathcal{C}$ by filling all the holes of $\mathcal{C}$ with the same M. We also write $\mathcal{C}\{N/x\}$ for the context obtained from the capture free substitution of N for x in $\mathcal{C}$. In particular, $\mathcal{C}\{N/x\}[M_i]_i$ denotes the term obtained by filling each hole $[\,]_i$ of the context $\mathcal{C}\{N/x\}$ with M_i.

A *tail context*, ranged over by $\mathcal{T}$, is defined by the following syntax.

$$\begin{array}{rcl} \mathcal{T} & ::= & [\,] \\ & \mid & \texttt{case}\ M\ \texttt{of}\ p\texttt{=>}\mathcal{T},\ldots,p\texttt{=>}\mathcal{T} \\ & \mid & \texttt{let}\ x = M\ \texttt{in}\ \mathcal{T}\ \texttt{end} \end{array}$$

A tail context $\mathcal{T}$ is a term with "holes" at its all tail positions. We write $\mathcal{T}[M_i]_i$ for the term obtained by filling each hole $[\,]_i$ in $\mathcal{T}$ with M_i.

3.2 Fusion laws for single parameter functions

We assume the standard call by name semantics. Later in Section 6, we shall comment on applying our fusion method to strict languages.

Our first fusion law is "apply distribution" defined below.

$$\text{(AppDist)}\quad f\ \mathcal{T}[M_i]_i \Longrightarrow \mathcal{T}[f\ M_i]_i$$

By this reduction, (static) evaluation can continue. This rule pulls out the head term of a case expression. For this rule to be sound, it is therefore necessary for f to be strict. Note that this condition is enforced in short-cut fusion.

THEOREM 1. *(AppDist) is sound for any strict f.*

Proof. Here we assume a standard environment-style denotational semantics $[\![M]\!]\eta$ of lambda term M under environment η, and show the semantic equation:

$$[\![f\ \mathcal{T}[M_i]_i]\!]\eta = [\![\mathcal{T}[f\ M_i]_i]\!]\eta$$

The following argument does not depend on specific property of the semantic definition.

The proof is by simple induction on the structure of the tail context $\mathcal{T}$. The base case is the identity. Suppose $\mathcal{T} = \texttt{case}\ M_0\ \texttt{of}\ \cdots, p_k\ \texttt{=>}\ \mathcal{T}_k, \cdots$. Then

$$\mathcal{T}[f\ M_i]_i = \texttt{case}\ M_0\ \texttt{of}\ \cdots, p_k\ \texttt{=>}\ \mathcal{T}_k[f\ M_{j_k}]_{j_k}, \cdots$$

Suppose $[\![f\ \mathcal{T}[M_i]_i]\!]\eta \neq \bot$. Since f is strict, $[\![\mathcal{T}[M_i]_i]\!]\eta \neq \bot$. Suppose $[\![M_0]\!]\eta = C_i(v_1, \ldots, v_n)$ for some C_i such that $p_i = C_i(x_1, \ldots, x_n)$. Then we have the following.

$$\begin{array}{rcl}
[\![f\ (\texttt{case}\ M_0\ \texttt{of}\ \cdots, p_i\ \texttt{=>}\ \mathcal{T}_i[M_{j_i}]_{j_i}, \cdots)]\!]\eta \\
\quad = & [\![f\ \mathcal{T}_i[M_{j_i}]_{j_i}]\!]\eta \cup \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \\
\quad = & [\![\mathcal{T}_i[f\ M_{j_i}]_{j_i}]\!]\eta \cup \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \\
& \text{(by induction hypothesis)} \\
\quad = & [\![\texttt{case}\ M_0\ \texttt{of}\ \cdots, p_i\ \texttt{=>}\ \mathcal{T}_i[f\ M_{j_i}]_{j_i}, \cdots]\!]\eta
\end{array}$$

The cases other than $[\![M_0]\!]\eta \neq C_i(v_1, \ldots, v_n)$ are trivial in an untyped semantics, or vacuous in a typed semantics.

The case for $\texttt{let}\ x = M\ \texttt{in}\ \mathcal{T}\ \texttt{end}$ is simpler and omitted. □

Our main law for fixed point promotion is the following inference rule.

$$\text{(FixPromote)}\quad \frac{f \circ M \stackrel{+}{\Longrightarrow} \mathcal{C}[f \circ g]}{f \circ \texttt{fix}\, g.M \Longrightarrow \texttt{fix}\, h.(\mathcal{C}\{\texttt{fix}\, g.M/g\})[h]}$$

where $\stackrel{+}{\Longrightarrow}$ represents one or more applications of the transformation relation $\Longrightarrow$ generated by our fusion laws (together with standard simplifying reduction rules.)

To show the soundness of this rule, we first show a general equational property. We write $M = M'$ for the β equality relation with the following rule for fix

$$\texttt{fix}\, g.M = M\{\texttt{fix}\, g.M/g\}$$

and we write $M \equiv N$ for the syntactic and definitional equality.

LEMMA 1. *If $f \circ M = \mathcal{C}[f \circ g]$ then $f \circ \texttt{fix}\, g.M$ is a fixed point of the functional $\lambda h.(\mathcal{C}\{\texttt{fix}\, g.M/g\})[h]$.*

Proof. Assume $f \circ M = \mathcal{C}[f \circ g]$. We have the following.

$$\begin{array}{rcl}
f \circ \texttt{fix}\, g.M & \equiv & \lambda x.f((\texttt{fix}\, g.M)\ x) \\
& = & \lambda x.f((M\{\texttt{fix}\, g.M/g\})\ x) \\
& \equiv & \lambda x.(f(M\ x))\{\texttt{fix}\, g.M/g\} \\
& \equiv & (f \circ M)\{\texttt{fix}\, g.M/g\} \\
& = & \mathcal{C}[f \circ g]\{\texttt{fix}\, g.M/g\} \text{ (by assumption)} \\
& = & \mathcal{C}[f \circ \texttt{fix}\, g.M]\{\texttt{fix}\, g.M/g\} \\
& \equiv & \mathcal{C}\{\texttt{fix}\, g.M/g\}[f \circ \texttt{fix}\, g.M] \\
& = & (\lambda h.\mathcal{C}\{\texttt{fix}\, g.M/g\}[h])(f \circ \texttt{fix}\, g.M) \quad \square
\end{array}$$

We further assume the standard denotational semantics based on the standard domain theory, where the fixed point construct denotes the least fixed point in the continuous function space.

THEOREM 2. *(FixPromote) is sound for any strict f.*

Proof. In the following argument, we implicitly identify terms with their denotations in the domain theory. Since the denotation of $\texttt{fix}\, h.(\mathcal{C}\{\texttt{fix}\, g.M/g\})[h]$ is the least fixed point, Lemma 1 establishes the following.

$$f \circ \texttt{fix}\, g.M \sqsupseteq \texttt{fix}\, h.(\mathcal{C}\{\texttt{fix}\, g.M/g\})[h]$$

To show the converse, let $F = \lambda w.\mathcal{C}\{\texttt{fix}\, g.M/g\}[w]$, and write $F^n(x)$ for $\underbrace{F(F \cdots (F}_{n \text{ times}}\ x) \cdots)$. We first show the following inequation for all n by induction on n.

$$F^n(\bot) \sqsupseteq f \circ ((\lambda g.M)^n(\bot))$$

This holds for $n = 0$ since $f \circ \bot = \bot$. The induction step is as follows.

$$\begin{array}{rcl}
F^{n+1}(\bot) & = & \mathcal{C}\{\texttt{fix}\, g.M/g\}[F^n(\bot)] \\
& \sqsupseteq & \mathcal{C}\{\texttt{fix}\, g.M/g\}[f \circ ((\lambda g.M)^n(\bot))] \\
& \sqsupseteq & \mathcal{C}\{(\lambda g.M)^n(\bot)/g\}[f \circ ((\lambda g.M)^n(\bot))] \\
& = & \mathcal{C}[f \circ g]\{(\lambda g.M)^n(\bot)/g\} \\
& = & (f \circ M)\{(\lambda g.M)^n(\bot)/g\} \\
& = & f \circ ((\lambda g.M)((\lambda g.M)^n(\bot))) \\
& = & f \circ ((\lambda g.M)^{n+1}(\bot))
\end{array}$$

From the above inequality, we have the following.

$$\begin{array}{rcl}
\texttt{fix}\, h.(\mathcal{C}\{\texttt{fix}\, g.M/g\})[h] & = & \sqcup \{F^n(\bot) \mid n \geq 0\} \\
& \sqsupseteq & \sqcup \{f \circ (\lambda g.M)^n(\bot) \mid n \geq 0\} \\
& = & f \circ \sqcup \{(\lambda g.M)^n(\bot) \mid n \geq 0\} \\
& & \text{(since } f \text{ and } \circ \text{ are continuous)} \\
& = & f \circ \texttt{fix}\, g.M
\end{array}$$

This concludes the proof. □

3.3 Fusion laws for multiple parameter functions

Both of (AppDist) and (FixPromote) extend systematically to multi-parameter functions. The case where the inner g is a multi-parameter function is straightforward. Here we only show the following two cases.

1. The outer f is a two parameter curried function whose first argument is the target of fusion.
2. The outer f is a two parameter uncurried function whose first argument is the target of fusion.

For the curried case, the law (AppDist) and (FixPromote) become (AppDist-c) and (FixPromote-c) as shown in Figure 1. For these laws, we can show the following.

THEOREM 3. *(AppDist-c) is sound for any function f that is strict with respect to the first argument, i.e. $f\ \bot\ y = \bot$.*

LEMMA 2. *If $f\ (M\ x)\ y = \mathcal{C}[f\ (g\ M_1^i)\ M_2^i]_i$ then*

$$\lambda x.\lambda y.f((\texttt{fix}\, g.M)\ x)\ y$$

is a fixed point of $\lambda h.\lambda x.\lambda y.\mathcal{C}[h\ M_1^i\ M_2^i]_i\{\texttt{fix}\, g.M/g\}$.

For the uncurried case, the law (AppDist) and (FixPromote) become (AppDist-u) and (FixPromote-u) as shown in Figure 1. In these definitions, we have used extended lambda terms: $\lambda(x, y).M$ for two-parameter lambda abstraction and $M(M_1, M_2)$ for two-parameter lambda application. For these laws, we can show the following.

THEOREM 4. *(AppDist-u) is sound for any function f that is strict with respect to the first argument, i.e. $f\ (\bot, y) = \bot$.*

$$\text{(AppDist-c)}\quad f\ \mathcal{T}[M_i]_i\ x \Longrightarrow \mathcal{T}[f\ M_i\ x]_i$$

$$\text{(FixPromote-c)}\quad \frac{f\ (M\ x)\ y \Longrightarrow \mathcal{C}[f\ (g\ M_1^i)\ M_2^i]_i}{\lambda x.\lambda y.f\ ((\texttt{fix}\, g.M)\ x)\ y \Longrightarrow \texttt{fix}\, h.\lambda x.\lambda y.\mathcal{C}[h\ M_1^i\ M_2^i]_i\{\texttt{fix}\, g.M/g\}}$$

$$\text{(AppDist-u)}\quad f\ (\mathcal{T}[M_i]_i, x) \Longrightarrow \mathcal{T}[(f\ M_i, x)]_i$$

$$\text{(FixPromote-u)}\quad \frac{f\ (M\ x, y) \Longrightarrow \mathcal{C}[f\ (g\ M_1^i, M_2^i)]_i}{\lambda(x,y).f\ ((\texttt{fix}\, g.M)\ x, y) \Longrightarrow \texttt{fix}\, h.\lambda(x,y).\mathcal{C}[h\ (M_1^i, M_2^i)]_i\{\texttt{fix}\, g.M/g\}}$$

Figure 1. Fusion Laws extended to Multi-parameter Functions

LEMMA 3. *If* $f\ (M\ x, y) = \mathcal{C}[f\ (g\ M_1^i, M_2^i)]_i$ *then*

$$\lambda(x,y).f\ ((\texttt{fix}\, g.M)\ x, y)$$

is a fixed point of $\lambda h.\lambda(x,y).\mathcal{C}[h\ (M_1^i, M_2^i)]_i\{\texttt{fix}\, g.M/g\}$.

Theorems 3, 4 and Lemmas 2, 3 can be proved similarly to the corresponding proofs of Theorem 1 and Lemma 1. We expect that semantic correctness of these generalized lows can be shown similarly to the proof of Theorem 2.

Note that it is straightforward to generalize these results to the cases with functions having more than two parameters. Our fusion algorithm shown in Section 4 and our implementation described in Section 6 treat general multi-parameter cases.

4. The Lightweight Fusion Algorithm

We now develop a lightweight fusion algorithm by embedding the fusion mechanism explained above in an inlining process. The strategy is summarized below.

- The algorithm recursively evaluates the input term using a *binding environment*.
- In order to suppress repeated fusion computations, the algorithm also maintains a *fusion environment* that records the results of previous fusion attempts for pairs of variables that are bound to recursive functions.
- When it encounters a function composition $f \circ g$, the algorithm first checks in the binding environment whether both f and g are bound to recursive functions. If this is the case then the algorithm performs the following actions depending on the past history of the pair (f, g) recorded in the fusion environment.
  - If fusion for this pair has succeeded before, then it returns the new function name f_g that was created by the previous fusion attempt and recorded in the fusion environment.
  - If fusion failed before, then it returns $f \circ g$.
  - If fusion has never been tried, then the algorithm attempts to fuse the two by applying the fusion laws explained in Section 3, and returns the environment extended with the result of the fusion attempt.
- When it encounters a term `let` h `=` M_1 `in` M_2 `end`, the algorithm processes M_1 and obtains the new terms, and then processes M_2 with the updated fusion environment. The algorithm then checks whether the fusion environment holds new function definitions that should be inserted into this let binding according to the following policy. If functions f and g are defined in this order, and $f\_g = M$ is the new function definition created by fusing f and g, then f_g is inserted immediately after the binding of g.

  In a strict language, a further special treatment of bindings is necessary during the computation of the (FixPromote) rule. We shall discuss this issue in Section 6.3.
- For all the other terms, the algorithm performs standard inlining and simplification.

To define the algorithm, we introduce some notations. We use f, g for variables bound to functions. We let μ range over binding environments, which is a mapping from variables to terms. We write $\mu\{x \mapsto M\}$ for the environment obtained from μ by adding the binding $\{x \mapsto M\}$, and write $\mu|_{\overline{x}}$ for the environment obtained from μ by deleting the entry of x. We let η range over fusion environments, which is a mapping from pairs of variables to one of the following results of fusion attempts.

- *Undefined* : fusion has never been tried for the pair.
- *Failed* : fusion has failed for this pair.
- *Succeeded* M: fusion has succeeded for the pair with the fused function term M.
- *Inserted* : fusion has succeeded and the binding has already been inserted as a binding in some let expression.

We write $\eta\{(f,g) \mapsto M\}$ for the environment obtained from η by adding the binding $\{(f,g) \mapsto M\}$. Under the bound variable convention for terms, both a variable and a pair of variables are globally unique and therefore $\mu\{x \mapsto M\}$ and $\eta\{(f,g) \mapsto M\}$ are always well defined. We adopt the convention that the new fused function name for a pair $f \circ g$ is f_g (determined from f, g), and we do not record this name in a fusion environment.

The lightweight fusion algorithm is now defined as a function $\mathcal{F}[\![\_]\!]$ of the following functionality.

$$\mathcal{F}[\![M]\!]\ \eta\ \mu = (M', \eta')$$

In order to avoid notational complication, we give the definition of $\mathcal{F}[\![\_]\!]$ for one-parameter functions, and explain the necessary extensions for the multi-parameter case later.

The algorithm is given in Figure 2 and 3. It only shows the cases relevant to fusion. The other cases are simple recursive evaluation as in ordinary inlining. The algorithm uses the sub-algorithms *fusion* and *dist*. The function *fusion* is for fusing two recursive functions; *fusion* $(f, \lambda x.M_f)\ (g, \lambda x.M_g)\ \mu\ \eta$ tries to fuse f and g in the expression $\lambda x.f\ (g\ x)$ under the bindings $\{f \mapsto \texttt{fix}\, f.\lambda x.M_f, g \mapsto \texttt{fix}\, g.\lambda x.M_g\}$ according to the method described in Section 3. The substitution $\{\lambda x.M_f/f\}$ used in the definition of *fusion* indicates that f is inlined once in $\mathcal{F}[\![\_]\!]$. The pattern $\mathcal{C}[f\ (g\ M_i)]_i$ used in case branches in *fusion* indicates that a term of the form $f\ (g\ M)$ is replaced by $f\_g\ M$ each time $\mathcal{F}[\![\_]\!]$ encounters the successive application of f and g. These can be implemented by adding some extra information in the entries of the fusion environment μ. *dist*(f, M) distributes the function variable f to all the tail positions in M according to the definition given in Section 3.

4.1 Extensions to multiple parameter functions

We describe the necessary extensions to deal with multiple parameter functions. Here we only consider uncurried functions, since an

$\mathcal{F}[\![M]\!]\ \eta\ \mu$ =
  case M of
    $(f\ (g\ M_0)) \Rightarrow$
      let $(M_0', \eta') = \mathcal{F}[\![M_0]\!]\ \eta\ \mu$
      in case $\eta'((f,g))$ of
          Failed $\Rightarrow$ $((f\ (g\ M_0')),\ \eta')$
        | *Succeeded* _ $\Rightarrow$ $(f\_g\ M_0',\ \eta')$
        | *Inserted* $\Rightarrow$ $(f\_g\ M_0',\ \eta')$
        | *Undefined* $\Rightarrow$
            if $\mu(g) = \texttt{fix}\, g.\lambda x.M_g$ and
               $\mu(f) = \texttt{fix}\, f.\lambda x.M_f$
            then
              let
                $\eta'' =$ *fusion* $(f,\ \lambda x.M_f)\ (g,\ \lambda x.M_g)$
                                $\eta'\ \mu|_{\overline{g}}$
              in
                case $\eta''((f,g))$ of
                    Succeeded _ $\Rightarrow$ $(f\_g\ M_0',\ \eta'')$
                  | *Failed* $\Rightarrow$ $(f\ (g\ M_0'),\ \eta'')$
                end
            else we perform ordinary inlining
    end
  | `let` $h = M_1$ `in` M_2 `end` $\Rightarrow$
      let
        $(M_1', \eta_1) = \mathcal{F}[\![M_1]\!]\ \eta\ \mu$
        $(M_2', \eta_2) = \mathcal{F}[\![M_2]\!]\ \eta_1\ \mu\{h \mapsto M_1'\}$
      in
        if $\eta_2 = \eta_2'\{(f,g) \mapsto$ *Succeeded* $M_3)\}$ and
           ($f = h$ or $g = h$)
        then (`let` $h = M_1'$ `in let` $f\_g = M_3$ `in` M_2' `end`,
              $\eta_2'\{(f,g) \mapsto$ *Inserted*$\}$)
        else (`let` $h = M_1'$ `in` M_2' `end`, η_2)
      end
  | in all the other cases we perform ordinary inlining

fusion $(f, \lambda x.M_f)\ (g,\ \lambda x.M_g)\ \eta\ \mu =$
  let
    $M_{body} = dist(f, M_g)$
  in
    case M_{body} of
      $\mathcal{C}[f\ (g\ M_i)]_i \Rightarrow$
        let
          $(M_1, \eta') = \mathcal{F}[\![\mathcal{C}[f\_g\ M_i]_i\{\lambda x.M_f/f\}]\!]\ \eta\ \mu|_{\overline{f}}$
        in
          case M_1 of
              $\mathcal{C}[f\ (g\ M_i)]_i \Rightarrow$
                $\eta'\{(f,g) \mapsto$
                  Succeeded (`fix` $f\_g.\lambda x.\mathcal{C}[f\_g\ M_i]_i$)$\}$
            | _ $\Rightarrow$
                $\eta'\{(f,g) \mapsto$
                  Succeeded (`fix` $f\_g.\lambda x.M_1$)$\}$
          end
      | _ $\Rightarrow$
        case $\mathcal{F}[\![M_{body}\{\lambda x.M_f/f\}]\!]\ \eta\ \mu|_{\overline{f}}$ of
            $(\mathcal{C}[f\ (g\ M_i)]_i, \eta') \Rightarrow$
              $\eta'\{(f,g) \mapsto$ *Succeeded* (`fix` $f\_g.\lambda x.\mathcal{C}[f\_g\ M_i]_i$)$\}$
          | $(\_, \eta') \Rightarrow$
              $\eta'\{(f,g) \mapsto$ *Failed*$\}$
  end

Figure 2. Fusion algorithm $\mathcal{F}[\![\_]\!]$

$dist\ (f, M)$ =
  case M of
      `case` M_0 `of` p_1 `=>` $M_1,\ \ldots,\ p_n$ `=>` M_n
      $\Rightarrow$
      `case` M_0 `of`
        p_1 `=>` $dist\ (f, M_1),\ \ldots,\ p_n$ `=>` $dist\ (f, M_n)$
    | `let` $x = M_1$ `in` M_2 `end`
      $\Rightarrow$ `let` $x = M_1$ `in` $dist\ (f, M_2)$ `end`
    | _ $\Rightarrow$ $f\ M$

Figure 3. Distribution function $dist$

intermediate language of an optimizing compiler often uses the uncurried representation. This is also the case in our implementation.

The set of terms is extended to include the uncurried multiple parameter functions as follows.

$$M \quad ::= \quad \cdots \mid \lambda(x_1, \ldots, x_k).\ M \mid M\ (M, \ldots, M)$$

In what follows, we write $\vec{M}$ for a sequences of terms of the form $M_1, \ldots, M_k$ separated by comma. Similar notation is used for patterns. For example, $(\vec{M^1}, \ldots, \vec{M^n})$ represents a tuple term of the form $(M_1^1, \ldots, M_{k_1}^1, \ldots, M_1^n, \ldots, M_{k_n}^n)$.

The extended algorithm $\mathcal{FM}[\![\_]\!]$ processes multi-parameter functions as follows.

$\mathcal{FM}[\![M]\!]\ \eta\ \mu$ =
  case M of
    $f\ (\vec{M}) \Rightarrow$
    case $divide\ (\vec{M})$ of
      $(\vec{M_l}, g\ (\vec{M_m}), \vec{M_r}) \Rightarrow$
        $\ldots$
        let
          $\eta'' =$ *fusionM* $(f, \lambda(\vec{x}).M_f)\ (g, \lambda(\vec{x}).M_g)\ \eta'\ \mu|_{\overline{g}}$
        in
          $\ldots$

When the algorithm detects an application $f(\vec{M})$, $divide\ (\vec{M})$ searches for an application term of the form $g\ (\vec{M_m})$ in $\vec{M}$ and returns $(\vec{M_l}, g\ (\vec{M_m}), \vec{M_r})$ indicating that $\vec{M}$ are divided into three parts, the middle of which is the application term $g\ (\vec{M_m})$. The matched terms $\vec{M_l}, \vec{M_m}, \vec{M_r}$ are recursively processed by $\mathcal{FM}[\![\_]\!]$ and then used as part of the result as in the algorithm $\mathcal{F}[\![\_]\!]$. When f and g are recursive functions and they have not been tried to fuse, the extended fusion function *fusionM* is invoked and it tries to fuse f and g in the expression $\lambda(\vec{x_l}, \vec{x_m}, \vec{x_r}).\ f\ (\vec{x_l}, g\ (\vec{x_m}), \vec{x_r})$ under the bindings $\{f \mapsto \texttt{fix}\, f.\lambda(\vec{x}).M_f, g \mapsto \texttt{fix}\, g.\lambda(\vec{x}).M_g\}$. Note that we allow both the outer function f and the inner function g to have more than one arguments. The fusing function *fusionM* is obtained by refining *fusion* based on the rule (FixPromote-u) in Section 3. The sub-algorithm *dist* is also refined to *distM* according to (AppDist-u) rule.

distM $(f, (\vec{M_l}), M, (\vec{M_r}))$ =
  case M of
    `case` M_0 `of` p_1 `=>` M_1`,` $\ldots$`,` p_n `=>` M_n
      $\Rightarrow$ `case` M_0 `of`
          p_1 `=>` *distM* $(f, (\vec{M_l}), M_1, (\vec{M_r})), \ldots,$
          p_n `=>` *distM* $(f, (\vec{M_l}), M_n, (\vec{M_r}))$
  | `let` x_1 `=` M_1`,` $\ldots$`,` x_k `=` M_k `in` M_0 `end`
      $\Rightarrow$ `let` x_1 `=` M_1`,` $\ldots$`,` x_k `=` M_k
        `in` *distM* $(f, (\vec{M_l}), M_0, (\vec{M_r}))$ `end`
  | _ $\Rightarrow f(\vec{M_l}, M, \vec{M_r})$

The arguments f, $\vec{M}_l$, $\vec{M}_r$ are always variables since the function *distM* is invoked in the function *fusion* as follows.

$$\text{let } M_{body} = \textit{distM}\ (f,\ (\vec{x_l}), M_g, (\vec{x_r})) \text{ in } \ldots$$

So the function *distM* does not cause code duplication.

The function *fusionM* tries to find successive function applications of the form $f\ (\vec{M_l^i}, g\ M_i, \vec{M_r^i})$ and replaces them with $f\_g\ (\vec{M_l^i}, M_i, M_r^i)$. By using the pattern matching notation as in the algorithm $\mathcal{F}[\![\_]\!]$, we can write the matching part as follows:

$$\begin{array}{l}\text{case } M_{body} \text{ of}\\ \quad \mathcal{C}[f\ (\vec{M_l^i}, g\ M_i, \vec{M_r^i})]_i \Rightarrow \ldots\end{array}$$

where $\vec{M_l^i}$ and $\vec{M_r^i}$ are pattern variables used to match the arguments. When the matching succeeds, the algorithm creates the term of the form $\mathcal{C}[f\_g\ (\vec{M_l^i}, M_i, M_r^i)]_i$. This achieves the replacement of the composition $f(\vec{M_l^i}, g\ M_i, \vec{M_r^i})$ of f and g with uncurried multi-parameters by the application $f\_g\ (\vec{M_l^i}, M_i, M_r^i)$ of new function f_g to the multiple parameters.

5. Examples

We claim that the lightweight fusion method we have just presented has practical significance in two ways. Firstly, it is readily implementable in any optimizing compiler that performs inlining, and it does not produce extra runtime overhead (except for code size increase due to the fact that both fused functions and the original functions are referenced.) Secondly, although it does not have any formal completeness property, it is powerful enough to fuse a wide range of recursively defined functions.

As we shall report in Section 6 and Section 7, we substantiate the first property by developing a simple inliner for a prototype compiler of a full scale functional language and testing it with some benchmarks. The second property can only be substantiated through long-term practical testing. We demonstrate its feasibility through examples that have been discussed in the literature.

5.1 Successful examples and their performance

This subsection shows typical examples that our lightweight fusion algorithm can deal with. All of them are successfully fused by our implementation without any preparation or ad-hoc modification to the algorithm. We measured execution time, heap usage, and file size for the examples with or without the inlining and fusion on our abstract machine. The input data are lists with 10^5 elements and trees with 2^{18} elements. Tables 1, 2, and 3 show the performance results. The meaning of the column marks are: "-" is without inlining, "**I**nline" is with inlining, "**F**usion" is with inlining and fusion, and "**F/I**" is the ratio of the last two.

As we shall describe in Section 6, our implementation is in a compiler for SML#, an extension of Standard ML. So let us use Standard ML syntax in showing examples. Although fusion is done for a typed polymorphic intermediate language, in showing examples, we omit type information to avoid cluttering the programs.

- Sum of squares of a list of integers.
  This is the old friend of fusion, whose fusion steps have been shown in Section 2.

  Before fusion:

```
let fun sum [] = 0
      | sum (x::xs) = x + sum xs
    fun mapsq [] = []
      | mapsq (x::xs) = x*x :: mapsq xs
in sum (mapsq list)
end
```

  After fusion:

```
let fun sum_mapsq [] = 0
      | sum_mapsq (x::xs) = x*x + sum_mapsq xs
in sum_mapsq list
end
```

- Sum of the integers from n to m.
  An integer list is produced by `from` and is consumed by `sum`.

  Before fusion:

```
let fun sum [] = 0
      | sum (x::xs) = x + sum xs
    fun from a b = if a > b then []
                   else a :: from (a+1) b
in sum (from 100000 200000)
end
```

  After fusion:

```
let fun sum_from a b = if a > b then 0
                       else a + sum_from (a+1) b
in sum_from 100000 200000 end
```

- `foldl`: Tail recursive sum of squares of list of integers.
  This is a rewrite of the previous example using `foldl`. The tail recursive `foldl` is much preferred to `foldr` if the result is the same. There seems to be no practically implemented automated fusion system that can deal with this simple example.

  Before fusion:

```
let fun from a b = if a > b then []
                   else a :: from (a+1) b
    fun foldl f r [] = r
      | foldl f r (x::xs) = foldl f (f (x,r)) xs
in foldl (op +) 0 (from 100000 200000)
end
```

  After fusion:

```
let fun foldl_from f r a b =
       if a > b then r
       else foldl_from f (f (a,r)) (a+1) b
in foldl_from (op +) 0 100000 200000 end
```

- Fusion of map.
  A boolean function `even` is applied to each element in the input integer list. Then the function `allTrue` checks whether all the obtained values are true or not.

  Before fusion:

```
let fun allTrue [] = true
      | allTrue (x::xs) = x andalso (allTrue xs)
    fun map f [] = []
      | map f (x::xs) = f x :: map f xs
    fun even x = x mod 2 = 0
in allTrue (map even list)
end
```

  After fusion:

```
let fun allTrue_map f [] = true
      | allTrue_map f (x::xs) =
          f x andalso (allTrue_map f xs)
    fun even x = x mod 2 = 0
in  allTrue_map even list
end
```

Table 1. Execution time for typical examples (in seconds)

Program	–	**I**nline	**F**usion	**F/I**
sum_mapsq	0.27	0.27	0.21	78%
sum_from	0.21	0.22	0.15	68%
foldl_from	0.27	0.27	0.15	56%
allTrue_map	0.43	0.33	0.18	55%
sumTree_mapsqTree	0.77	0.76	0.51	67%

Table 2. Heap allocation for typical examples (in the number of fields divided by one thousand)

Program	–	**I**nline	**F**usion	**F/I**
sum_mapsq	600.2	600.2	300.2	50.0%
sum_from	300.2	300.2	0.2	0.1%
foldl_from	500.2	500.2	200.2	40.0%
allTrue_map	1,200.3	600.2	300.2	50.0%
sumTree_mapsqTree	2,621.7	2,621.7	1,310.9	50.0%

In this case, the function `even` is simply passed to the next function call without any change. This is just an extra parameter but not the so called accumulating parameter. Our fusion algorithm does not depend on whether the parameter is accumulating or not.

- Fusion of functions on trees
  Our fusion does not depend on lists. In fact, our fusion system does not even know any property of list. Our system can fuse functions on user defined tree-like structure. Here is a simple example of fusing tree.

  Before fusion:

```
let fun sumTree Empty = 0
      | sumTree (Node (x,t1,t2)) =
        x + sumTree t1 + sumTree t2
    fun mapsqTree Empty = Empty
      | mapsqTree (Node (x,t1,t2)) =
        Node (x * x,
              mapsqTree t1,
              mapsqTree t2)
in sumTree (mapsqTree tree)
end
```

  After fusion:

```
let fun sumTree_mapsqTree t =
        case t of
          Empty => 0
        | Node (x,t1,t2) =>
          x * x
          + sumTree_mapsqTree t1
          + sumTree_mapsqTree t2
in sumTree_mapsqTree tree
end
```

5.2 Examples that cannot be fused

As shown in the above, our fusion algorithm works on typical examples, but of course it does not fuse all the fusable functions. We give two examples below to show the limitation of our current fusion algorithm.

Table 3. Program sizes for typical examples (in byte)

Program	–	**I**nline	**F**usion	**F/I**
sum_mapsq	3,567	3,371	3,459	102.6%
sum_from	2,278	2,126	2,174	102.3%
foldl_from	3,354	3,274	2,546	77.8%
allTrue_map	5,011	4,599	4,543	98.8%
sumTree_mapsqTree	5,420	4,556	4,596	100.9%

- Two successive applications of the same function.
  Our algorithm does not fuse two successive applications of the same function like the following.

```
let fun mapsq [] = []
      | mapsq (x::xs) = x*x :: mapsq xs
in mapsq (mapsq list)
end
```

  This is because our fusion algorithm performs inlining of each function only once. We firstly inline the inner application of `mapsq`. Then we distribute the outer `mapsq` to the body of inner `mapsq`. After the distribution the fusion algorithm tries to inline the outer `mapsq`, but it fails since (the inner) `mapsq` has already been inlined.

- Mutually recursive functions.
  Our current algorithm does not fuse mutually recursive functions such as the following.

```
let fun f [] = []
      | f (x::xs) = 2 * x :: g xs
    and g [] = []
      | g (x::xs) = 3 * x :: f xs
    fun sum [] = 0
      | sum (x::xs) = x + sum xs
in sum (f list)
end
```

  This limitation is due to our simple strategy to treat a pair of functions (f, g) as a unit of fusion, and to inline each of the two functions only once during each fusion trial. We believe it possible to refine this strategy to deal with mutual recursion. For example, if g is one of mutually recursive set of functions $\{g_1, \ldots, g_n\}$, then one strategy is to try to fuse the set of pairs $\{(f, g_1), \ldots, (f, g_n)\}$ simultaneously, with the restriction that each function is inlined once for each pair. We leave the refinement to future work.

6. Implementation

We have implemented the lightweight fusion algorithm given in Section 4. This section describes the overview of the implementation and discusses several issues we had to cope with when implementing our algorithm.

For the reader to test our implementation or to read the code, we have made the implementation (including the source code) available through the Internet for downloading at: `http://www.pllab.riec.tohoku.ac.jp/software/fusion/index.html`.

6.1 Implementation in a strict language

The implementation is done by adding an inliner phase to a prototype compiler for SML# [20], which is an extension of Standard ML, an eager language having imperative features.

The rationale of choosing the SML# compiler is our intention to investigate the feasibility of program fusion in a strict and impure language in future. Although fusion has mainly been investigated

and implemented in lazy languages such as Haskell, the principle of eliminating redundant intermediate data structure is of course equally important in strict and impure languages. The major problem in applying fusion to imperative languages is the preservation of the order of imperative effects. As we shall examine in details in 6.3, fusion transformation in general changes the order of evaluation and thus its straightforward application to functions with side effects is unsound. We expect that, with a proper control of interaction between fusion and imperative features, fusion should become an important optimization for strict and impure languages as well. We note that this situation is analogous to the relationship between fusion and non-strict functions in lazy languages.

This issue is outside the scope of the present paper, and we would like to investigate it elsewhere. Our current implementation does not properly deal with imperative features. However, it is sufficient for the purpose of the present paper stated in the introduction section.

6.2 Implementing multiple argument fusion

Our fusion algorithm works on functions with multiple arguments. We have given the rules for curried and uncurried versions, and our method works on both versions. Among them, we have only implemented the uncurried version since the SML# compiler performs uncurry optimization, which transforms multi-parameter functions in curried form into uncurried intermediate code. For example, the map function is translated to the following term in the intermediate language.

```
['a,'b. val rec map =
 (fn {f, x} =>
   (case x of
      nil => nil {'b}
    | :: y => :: {'b}(f (#1 y), map {f, (#2 y)})
    | _ => raise MatchCompBug))]
```

where `{f, x}` represents the multiple parameters. Our fusion algorithm for functions with multiple parameters works on this form of terms.

6.3 Inlining bindings for function applications

In our implementation of the fusion algorithm, some special treatment of bindings is needed in addition to the usual inlining techniques. For the fusion of two functions f and g to succeed, successive function applications $f\ (g\ \_)$ should appear during the computation of the (FixPromote) rule for f and g. This implies that a binding of the form $x = g\ e$ needs to be inlined during the fusion of f and g despite the fact that $g\ e$ is not a value. The strategy we have implemented is to inline the binding of the form $x = g\ e$ if the following conditions are met: (1) the inliner is processing the fusion for $f \circ g$, and (2) x occurs in the context of $f\ x$ so that the inlining this binding will induce the replacement of $f \circ g$ with f_g. This means that successful fusion may change the order of evaluation if there is some computation between the binding $x = g\ e$ and the application $f\ x$.

Let us describe this strategy and the related issues by tracing the fusion process for a variant of `sum_mapsq` example, where the function `sum` is replaced with the following.

```
fun dsum [] = 0
  | dsum (x::xs) = 2 * x + dsum xs
```

The function `mapsq` is compiled to the following intermediate code.

```
val rec mapsq =
  (fn x =>
    (case x of
       nil => nil
     | :: y => bind x = #1 y
               in bind xs = #2 y
                  in ::(*(x,x), mapsq xs) end
               end
   | _ => raise MatchCompBug))
```

Firstly, we create a new function name `dsum_mapsq` and then distribute `dsum` into the body of `mapsq`. In the following, we omit the raise expression.

```
(fn x =>
  (case x of
     nil => dsum (nil)
   | :: y => bind x = #1 y
             in bind xs = #2 y
                in dsum (::(*(x,x), mapsq xs))
                end
             end))
```

In the next step, we apply the main function, $\mathcal{FM}[\![\_]\!]$, to the obtained term. The case branches p_i `=>` e_i are processed one by one, from the top to the bottom. The first case is just inlining of `dsum` to the empty list, which results in `0`. Note that we allow inlining of recursive function once. In the next branch, we inline the bindings `bind x = #1 y` and `bind xs = #2 y`. At this point, the term becomes as follows.

```
(fn x =>
  (case x of
     nil => 0
   | :: y => dsum (::(*(#1 y, #1 y),
                      mapsq (#2 y)))))
```

We inline the application of `dsum` to the cons list. When inlining a function applied to a term of some constructor, we make a binding for each argument of the constructor or put it in the inliner environment, depending on whether or not the argument is small and so on. In the usual inliner, the term at this point would become as follows.

```
(fn x =>
  (case x of
     nil => 0
   | :: y => bind z = *(#1 y, #1 y)
             in bind zs = mapsq (#2 y)
                in +(*(2,z), dsum zs)
                end
             end))
```

However, this does not make the successive application of `dsum` and `mapsq`. So, we allow all the function applications to be put into the inliner environment with the tag indicating that this should only be inlined when the inlining makes fusion succeed. In this case inlining `zs` makes the successive application, so we inline `zs`.

```
(fn x =>
  (case x of
     nil => 0
   | :: y => bind z = *(#1 y, #1 y)
     in +(*(2,z), dsum (mapsq (#2 y))
     end))
```

Note that the evaluation order changes by the inlining of `zs` since a multiplication occurs immediately after the squaring of the head element. By replacing the successive applications of `dsum` and `mapsq` with `dsum_mapsq`, the evaluation order furthermore changes recursively.

```
(fn x =>
  (case x of
     nil => 0
```

```
    | :: y => bind z = *(#1 y, #1 y)
              in +(*(2,z), dsum_mapsq (#2 y))
              end))
```

This result is put into the fusion environment η. Our algorithm performs this sequence of steps. The following is the actual term generated by our compiler for the input term.

```
val rec dsum_mapsq =
  (fn $8 =>
    (case $8 of
       nil => 0
     | ::$19 =>
       bind newVar = *(#1 $19, #1 $19)
       in +(*(2,newVar), dsum_mapsq (#2 $19)) end
     | _ => raise MatchCompBug))
```

Note that an application of a function to a raise expression can be reduced to the same raise expression with its type appropriately changed.

Let us remark a related issue concerning both strict and lazy languages. The inlining of bindings for function applications may in general cause work duplication. In the above example, if `zs` occurs more than once in the scope of the binding `zs = mapsq (#2 y)`, work duplication occurs even if fusion transformation succeeds. This reflects the fact that our fusion method is for general recursive function definitions. There are various ways to cope with this situation under the trade off between the duplicated work and the effect of eliminating the production of data structures.

7. Benchmarks and Their Analysis

As we have stated in the introduction section, the purpose of our implementation is to substantiate the following claims:

- Our lightweight fusion algorithm can be readily implementable in an inlining phase of an optimizing compiler.
- It does not produce extra runtime overhead.

Our implementation and the performance results indeed verify these two points. Except for several special treatments described in the previous section, we were able to implement the algorithm in a compiler of a full scale language using usual inlining techniques. As shown in Section 5, the resulting system can fuse various typical examples discussed in the literature.

To verify the second point, we tested several standard benchmarks for Standard ML. We measured execution time, heap usage, and code sizes with or without inlining and fusion transformation. In the measurement, we made all the other optimizations on, including dead code elimination, uncurry optimization, constant folding, and tail call optimization. Tables 4, 5, and 6 show the benchmark results.

They show that our fusion transformation can be safely applied. When fusion has failed, it just does usual inlining without producing any runtime overhead. The heap usage and program size are the same as the case that fusion is off and inlining is on, except for the vliw benchmark. In the vliw benchmark, fusion occurs twice successfully, although the elimination of intermediate data structure does not affect the execution time so much. Heap usage and file size increase a little bit in the vliw benchmark when fusion option is on, since after fusing $(f \circ g)$ to f_g, the definitions of unused f or g may remain even after the dead code elimination. This is due to the limitation of our dead code elimination, which judges only direct death and does not check the transitive death.

Looking at these benchmark results in the perspective of the effectiveness of fusion method for general ML programs, they would give a pessimistic impression that fusion would not be effective in practice. However, we should note that the implementation and the

Table 4. Execution time for benchmarks (in seconds)

Benchmark	–	**I**nline	**F**usion	**F/I**
barneshut	21.0	16.7	16.7	100%
boyer	1.4	1.3	1.3	100%
coresml	89.6	88.6	88.4	100%
countgraphs	199.9	140.2	140.6	100%
fft	24.7	16.1	16.1	100%
knuthbendix	8.0	6.4	6.4	100%
lexgen	19.0	13.2	13.3	101%
life	2.5	2.6	2.6	100%
logic	34.1	26.3	26.4	100%
mandelbrot	13.7	12.3	12.3	100%
mlyacc	2.6	2.3	2.4	100%
nucleic	1.5	1.1	1.1	100%
ray	16.7	14.2	14.2	100%
simple	37.3	23.7	23.6	100%
tsp	8.5	7.3	7.3	100%
vliw	18.1	15.8	15.4	97%

Table 5. Heap allocation for benchmarks (in the number of fields divided by one thousand)

Benchmark	–	**I**nline	**F**usion	**F/I**
barneshut	102,515.4	94,234.4	94,234.4	100%
boyer	7,138.9	6,874.5	6,874.5	100%
coresml	0.3	0.3	0.3	100%
countgraphs	1,077,263.9	976,271.8	976,271.8	100%
fft	112,517.3	112,517.3	112,517.3	100%
knuthbendix	53,258.9	42,736.9	42,736.9	100%
lexgen	91,378.5	87,839.4	87,839.4	100%
life	4,313.5	3,832.9	3,832.9	100%
logic	180,964.5	119,571.6	119,571.6	100%
mandelbrot	69,444.4	65,241.9	65,241.9	100%
mlyacc	14,731.3	13,843.6	13,843.6	100%
nucleic	11,818.8	3,885.4	3,885.4	100%
ray	87,368.6	84,484.8	84,484.8	100%
simple	172,191.6	130,210.3	130,210.3	100%
tsp	30,313.4	19,851.9	19,851.9	100%
vliw	112,342.8	99,979.7	99,980.4	100%

Table 6. Code sizes for benchmarks (in kilo byte)

Benchmark	–	**I**nline	**F**usion	**F/I**
barneshut	205.4	231.7	231.7	100%
boyer	227.0	227.2	227.2	100%
coresml	1.7	1.5	1.5	100%
countgraphs	92.2	94.3	94.3	100%
fft	36.6	37.7	37.7	100%
knuthbendix	132.8	156.0	156.0	100%
lexgen	243.3	295.6	295.6	100%
life	51.3	62.6	62.6	100%
logic	144.8	154.8	154.8	100%
mandelbrot	8.4	8.3	8.3	100%
mlyacc	1436.8	1641.6	1641.6	100%
nucleic	636.2	583.3	583.3	100%
ray	95.6	104.4	104.4	100%
simple	263.5	347.7	347.7	100%
tsp	70.8	78.5	78.5	100%
vliw	794.6	941.3	944.9	100%

benchmarks reported in this section are quite preliminary; we performed them only for verifying the above two points. In particular, we put almost no effort to tune the fusion algorithm against the compiler and the language. As such, the preliminary results should not be interpreted as a definite negative evidence of the effectiveness of our light weight fusion method for strict imperative languages.

To analyze this issue, we have examined the fusion process of a few benchmarks and have identified the following three problems that have prevented fusion from occurring in our current implementation.

1. Our current implementation only fuses functions if they appear in a nested let expression. In particular, it cannot fuse any functions in different compilation units. Since functions in the basis library are always in different compilation units from that of the user program, they cannot be fused.
2. The current SML# compiler can only perform uncurrying optimization for locally defined functions. All the escaping functions are in curried form. This minimizes the opportunities for our multiple-parameter function fusion.
3. A function is often defined as a simple call to a locally defined recursive function. A typical case is a function that uses a tail recursive auxiliary function. Such a locally defined auxiliary recursive function is not recognized as a candidate of fusion.

The following code fragment in the "simple" benchmark illustrates the first two points.

```
flatten (map (fn k => (map (fn l => f (k,l))
                           (from(lmin,lmax))))
             (from (kmin,kmax)))
```

The function `map` is defined in the basis library and is in a curried form. The following code in the benchmark ray exhibits the third point.

```
implode (map fromStr x)
```

`implode` is defined in the basis library as follows.

```
fun implode chars =
  let
    fun scan [] accum = accum
      | scan (char :: chars) accum =
        scan chars (accum ^ (Char_toString char))
  in
      scan chars ""
  end
```

This has a locally defined recursive function inside and the function `implode` itself is not a recursive function.

The first problem can be solved by refining our environment management in such a way that the algorithm maintains an environment that keeps the body of each recursive function beyond its compilation unit. The second point requires either to implement the fusion rules for curried multi-parameter fusions or to improve the uncurrying optimization so that it works for non local definitions. The third point can be solved by inlining non-recursive (small) functions before fusion.

Remedying these three points and implementing a practical fusion algorithm are beyond the scope of the present paper. Instead, we have hand-simulated the necessary extensions for the "simple" benchmark by manually changing the source program. The result shows that fusion succeeds five times and the execution time decreases 5% compared with the case where the other optimizations, including inlining, are on. From this preliminary analysis, we expect that if we fine tune our method against the compiler and the language, our light weight fusion method would show reasonable speed up over already optimized code.

8. Related Work

In a general perspective, our fusion method can be regarded as a special case of the fold/unfold method of Burstall and Darlington [1]. A major problem of applying the general idea of fold/unfold transformation to automated program optimization is to control termination and to find a good strategy for folding (i.e. generating new recursive definitions). Due to this difficulty, there does not seem to exist an automatic fusion method that directly fuses general recursive definitions.

In the work of deforestation [23, 3], this problem was solved by restricting the class of fusable expressions. However, the restriction appears to be too strong for this method to be imporporated in a practical compiler. In short-cut fusion approach [6] or its extensions [7, 5], the problem was avoided by giving up fusing recursive definitions and instead developing a specific fusion laws for expressions using foldr and build. Warm fusion [12] and transformation to hylomorphism forms [9] also avoid this problem by transforming general recursion to specific forms to which short-cut fusion laws can apply. As we mentioned in the introduction, its practical feasibility remains to be seen.

Our method appears to be the first successful example that overcomes the difficulty of generating new recursive definition mentioned above and directly fuses general recursive definitions. The key insight of our approach is that a practical fusion system is obtained by focusing on a specific fusion pattern of recursive function definition and that this process is expressed by a combination of syntactic transformation rules without requiring any heuristics.

We share the motivation with the designers and developers of optimizing compilers with fusion transformation. One notable example is the GHC compiler's rules pragma [17], where the compiler is extended with a mechanism to specify general rewrite rules. This mechanism allows programmers, especially the designers of a basis library, to specify the short-cut fusion [6], its extension [7], array fusion [2], and many other useful optimizations for functions. Our system on the other hand allows fully automated fusion for general recursive function definitions.

There have been some attempts to implement warm fusion, including a program transformation system Stratego [10] and an experimental extension to the GHC compiler [15]. However, as reported in [15], there is some overhead problem because of the introduction of lambda abstractions. In [16], the GHC compiler is extended with the hylo fusion system [9]. Given that the underlying mechanism is more general than warm fusion [12] in the sense that it can deal with arbitrary data type, the benchmark results reported in [16] are quite encouraging. Considering the lack of a mechanism for functions with accumulating parameters such as tail recursive functions, its practical feasibility remains to be investigated.

Recent work by Katsumata and Nishimura [11] have shown that short-cut fusion can be generalized to incorporate accumulating parameters using algebraic properties of functions. This result can be used to extend automated fusion system such as the one [16] for wider range of recursive function definitions. But again, it remains to be seen whether a practically feasible automated optimization step can be extracted from its involved mathematical development.

Compared with those previous approaches, our method provides an alternative very light weight fusion method that can be incorporated to an optimizing compiler.

9. Conclusions and Future Work

We have developed a lightweight fusion method that automatically fuses recursive function definitions. The method is based on

a new fusion process, called fixed point promotion, that transforms $f \circ (\mathtt{fix}\, g.\lambda x.E)$ to a new fixed point term $\mathtt{fix}\, h.\lambda x.E'$ by promoting the function f through the fixed point operator. We have shown that this process is represented by the combination of two new transformation rules (AppDist) and (FixPromote) and have shown their soundness. We have then given a fusion algorithm by incorporating these rules in an inlining process. Since these rules are not sensitive to the types of functions being fused, our fusion method can be applicable to a wide range of functions including those with multi-parameters in both curried and uncurried form and also those that manipulate user-defined general data types. Since the two new laws are both simple and syntactic ones that only perform local transformation without requiring search or other heuristics, our method can be readily incorporated to any standard inlining optimization. To demonstrate the practical feasibility of our method, we have implemented our method in a compiler of a full-scale functional language, and have successfully tested various examples including those that cannot be dealt with in any existing real compiler. The performance result shows that our method does not introduce runtime overhead for large scale benchmarks.

This is our first step towards developing a practical lightweight fusion method, and a number of further issues remain to be investigated. The most important one is the evaluation of the feasibility of our method through a serious implementation. As we have noted, our current implementation is experimental and requires a number of refinements as discussed Section 7. Furthermore, our current implementation does not consider possible imperative effects and therefore it may not be sound for programs with side effects. We plan to combine our fusion method with a static effect system and include in our SML# compiler. Implementing our system in a lazy language such as GHC would also be a worthwhile future work.

Another interesting direction of research is to investigate various formal properties of our fusion lows. We have shown the semantic soundness using denotational semantics. One possible refinement wold be to adopt the idea of improvement theorem by David Sands [18] to our setting. Another important issue is the correctness in call-by-value semantics. Our correctness theorem (Theorem 2) depends on call-by-name semantics. A promising approach toward this direction would be to use the axioms for recursion in call-by-value by Hasegawa and Kakutani [8].

Acknowledgments

We are grateful to Masahito Hasegawa for his comments on the possibility of proving the correctness of our fusion laws based on their axioms for recursion in call-by-value. We would like to thank anonymous referees for many helpful suggestions and comments. Comments from Olaf Chitil, Philip Wadler, and Akimasa Morihata were useful in improving the paper.

References

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

[2] M. Chakravarty and G. Keller. Functional array fusion. In *Proc. ACM International Conference on Functional Programming*, pages 205–216, 2001.

[3] W-N. Chin. Safe fusion on functional expression. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 11–20, 1992.

[4] O. Chitil. Type inference builds a short cut to deforestation. In *Proc. ACM International Conference on Functional Programming*, pages 249–260, 1999.

[5] N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic augment and generalised short cut fusion. In *Proc. ACM International Conference on Functional Programming*, pages 294–305, 2005.

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

[7] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. Ph.D thesis, University of Glasgow, 1996.

[8] M. Hasegawa and Y. Kakutani. Axioms for recursion in call-by-value. *Higher-Order and Symbolic Computation*, 15(2-3):235–264, 2002.

[9] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proc. ACM International Conference on Functional Programming*, pages 73–82, 1996.

[10] P. Johann and E. Visser. Warm fusion in Stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):1 – 34, 2000.

[11] S. Katsumata and S. Nishimura. Algebraic fusion of functions with an accumulating parameter and its improvement. In *Proc. ACM International Conference on Functional Programming*, pages 227–238, 2006.

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

[13] G. Malcolm. Homomorphism and promotability. In *Proc. Mathematics of Program Construction*, pages 335–347, 1989.

[14] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. International Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, *Lecture Notes in Computer Science* 523, pages 124–144, 1991.

[15] L. Nemeth and S. Peyton Jones. A design for warm fusion. In *Proc. International Workshop on Implementation of Functional Languages*, pages 381–393, 1998.

[16] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. Verification of practical effectiveness of program fusion (in Japanese). *Computer Software*, 17(3):81–85, 2000.

[17] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proc. ACM Haskell workshop*, 2001.

[18] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1-2):193–233, 1996.

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

[20] SML# home page. http://www.riec.tohoku.ac.jp/smlsharp/, 2006.

[21] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proc. ACM International Conference on Functional Programming*, pages 124–132, 2002.

[22] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.

[23] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. Special issue of selected papers from 2'nd European Symposium on Programming.