Fusion with Stacks and Accumulating Parameters [*]

Susumu Nishimura
Department of Mathematics, Faculty of Science, Kyoto University, Kyoto 606–8502, Japan
susumu@math.kyoto-u.ac.jp

ABSTRACT

We propose a new algorithm for fusion transformation that allows both stacks and accumulating parameters. The new algorithm can fuse programs that cannot be handled by existing fusion techniques, *e.g.*, XML transformations. The algorithm is formulated in a modular, type-directed style where the transformation process is comprised of several transformation steps that change types but preserve the observational behavior of programs. We identify a class of functions to which our new fusion method successfully applies and show that a closure property holds for that class.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming—*program transformation*; D.1.1 [**Programming Techniques**]: Functional Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms

Algorithms, Languages

Keywords

Stacks, accumulating parameters, higher-order removal, shortcut fusion, type-directed program transformation.

1. INTRODUCTION

Modular programming, where a program is written as a collection of smaller parts that solve subproblems, makes programs significantly easier to understand and implement. However, finer modularity often goes against efficiency, because of the overhead of interfacing between subprograms. Functional programs, in which interfacing is explicit as function composition, are well-suited for studying this issue, and the research within the last two decades has developed a family of novel optimization techniques called *fusion* (or *deforestation*) [27, 4, 8, 20, 14, 22, 11]. Fusion is a program transformation which unifies two recursive functions into one. A remarkable improvement on efficiency is achieved when the two functions are interfaced by a large sized intermediate recursively defined data structures, such as lists.

In this paper, we propose a new fusion algorithm that can deal with *stacks* and *accumulating parameters*, which are difficult or even impossible to handle in the existing fusion methods. The new algorithm is intended to encompass more realistic applications. For example, consider a program for XML document transformation. Such a program would be implemented in the following modular style.

$$\text{XML} \xrightarrow{parse} \text{XML tree} \xrightarrow{f} \text{XML tree} \xrightarrow{unparse} \text{XML}$$

The input XML text is parsed by the function *parse* to produce an internal tree representation. Then, a function f for document transformation is applied to produce another intermediate tree representation for the output. The resulting tree is translated back into the text representation by *unparse* for the final output. In this program, stacks and accumulating parameters naturally arise. Stacks are needed in order to parse against non-regular languages [10]. Accumulating parameters are also required for propagating the context information during parsing, unparsing, and any other computation of a reasonable complexity. (The discussion above holds for many software systems that operate on non-regular languages, *e.g.*, compilers.)

The technical contributions of this paper are as follows.

- Our algorithm is formulated in a highly modular manner in the sense that the algorithm is comprised of a series of *type-directed* transformation steps, where each step represents a *type conversion* on the program. This type-directed formulation gives a logically clean separation of the transformation process into smaller pieces that are easier to handle. Higher-order functions are intensively used for interfacing between transformation steps. This differentiates our work from existing program composition techniques. (See the discussion in Section 6.)

- Our fusion algorithm always succeeds if the source program meets a *linearity condition.* It is also shown that a *closure property* holds, *i.e.*, if the source program meets a (stronger) linearity condition, the transforma-

[*]Part of preliminary result of this paper was informally presented in Asian Workshop on Programming Languages and Systems 2002 (APLAS'02).

tion result is also linear. This closure property guarantees that a program comprising of more than two functions can be fused by repeated applications of our algorithm.

In this paper, we discuss a fusion algorithm for two functions, each of which recurses over a single input data structure. (We do not consider zip-like functions that recurse over two or more inputs simultaneously.) We assume a call-by-name semantics for a functional language we consider. When we discuss fusion for a function composition $g \circ f$, we call f the *producer function* and g the *consumer function*, where the latter consumes the intermediate result produced by the former. Each function may have accumulating parameters, while only the producer function can have stacks in its accumulating parameters and/or return results for managing data elements in the LIFO order. Parts of final output may be held in accumulating parameters and stacks, whereas no inspection into these data elements (*e.g.*, case distinction) is allowed.

The rest of this paper is organized as follows. Section 2 gives our running example for which we demonstrate our fusion algorithm. In Section 3, we explain our fusion transformation in a simplified setting. Section 4 presents the full transformation algorithm, which extends the aforementioned algorithm with stacks and multiple accumulating parameters and return results. In Section 5, we provide a condition for successful transformation and show a closure property. Section 6 mentions related work and finally Section 7 concludes with a brief summary and some issues for future work.

For the lack of space, we omit proofs and some program codes from this paper. The omitted materials are available on-line from the author's homepage `http://www.math.kyoto-u.ac.jp/~susumu/`.

2. TRANSFORMATION EXAMPLE

As an example of fusion transformation, let us consider a program that operates on sequences of matching parentheses. We chose this simple example for explanatory purposes only. We can consider a more involved example, say, XML transformation systems [16, 18].

Throughout the paper, we present programs by borrowing the syntax from Haskell [9] and extending it with records and overloaded functions. (See Section 4 for the definition of overloaded functions.) We write $M[x_1 \backslash N_1, \ldots, x_n \backslash N_n]$ to represent a *substitution* to expression M, where the free occurrences of variables $x_1, \ldots, x_n$ in M are simultaneously replaced with $N_1, \ldots, N_n$, respectively. We also use a special symbol Ω to represent an arbitrary well-typed expression. Every occurrence of Ω represents an unreachable point in the program and hence can be replaced with any (appropriately typed) expression without affecting the program's behavior. In practice, we can encode Ω `= error "unreachable"`.

We model the sequence of (possibly unmatching) parentheses by the following datatype

```
data L = Begin L | End L | Nil
```

where the constructor `Begin` stands for the left parenthesis, `End` for the right parenthesis, and `Nil` for the end of sequence.

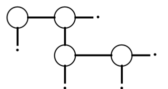
A sequence of matching parentheses is internally represented by a binary tree as defined below.

```
parse :: L → (se:: Stk T) → (l:: T, sp:: Stk T)
parse (Begin t) x0 =
  let x1 = parse t (se=Stk Leaf (#se x0))
      Stk a as = #sp x1
  in (l=Node (#l x1) a, sp=as)

parse (End t) x0 =
  let x1 = parse t (se=as)
      Stk a as = #se x0
  in (l=a, sp=Stk (#l x1) (#sp x1))

parse Nil x0 = (l=Leaf, sp=empty)

cascade :: T → (h:: T) → (l:: T)
cascade (Node t1 t2) x0 =
  let x1 = cascade t1 (h= Node (#l x2) Leaf)
      x2 = cascade t2 (h= #h x0)
  in (l= #l x1)

cascade Leaf x0 = (l= #h x0)

unparse :: T → (k:: L) → (j:: L)
unparse (Node t1 t2) u0 =
  let u1 = unparse t1 (k= End (#j u2))
      u2 = unparse t2 (k= #k u0)
  in (j= Begin (#j u1))

unparse Leaf u0 = (j= #k u0)
```

Figure 1: Source program

```
data T = Node T T | Leaf
```

Each binary node represents a matching pair of parentheses, where its left subtree corresponds to the subsequence embraced by that pair, while its right subtree corresponds to the rest of the sequence to follow. For example, a matching sequence ()(()()) is translated into a binary tree depicted below.



We consider a fusion for a system that is composed of three functions given in Fig. 1, where `parse` is the parsing function, `cascade` is the transformation function on the parsed binary tree, and `unparse` is the unparsing function.

The parsing function utilizes stacks, which we implement as follows.

```
data Stk a = Stk a (Stk a)
empty = error "empty stack"
```

We represent a stack with n elements $a_1, \ldots, a_n$ $(n \geq 0)$ by a list-like structure terminated by `empty`, *i.e.*, `Stk` a_1 `(Stk` a_2 `(` $\cdots$ `(Stk` a_n `empty)`$\cdots$`))`. In Fig. 1, we use pattern matching `Stk` a as `=` S to bind the topmost element of a stack S to the variable a and the rest to the variable as. This corresponds to the "pop" operation on stacks.

The functions in Fig. 1 are all defined as a recursive function that takes a set of accumulating parameters and returns a set of computation results, where each set is expressed by a *record*.[1] A record, written $(l_1 = M_1, \ldots, l_n = M_n)$

[1]The record syntax is available as `Trex` extension in Haskell interpreter Hugs (`http://www.haskell.org/hugs`).

($n \geq 1$), is a tuple whose each element M_i (often called a *record field*) is indexed by a unique *record label* l_i. We write $(l_1 :: \tau_1, \ldots, l_n :: \tau_n)$ for the type of records consisting of n fields where each record field labeled by l_i has type τ_i. (The order of labels is insignificant both in expressions and types.) To access a record field labeled by l in a record R, we use *field selection operator*, written $\#l\ R$.

We use record labels as a convenient means for marking certain set of expressions by syntactical objects. In the final transformation step of our algorithm (Section 3.4 and 4.6), it is crucial to identify some expressions according to their origin. Record labels are convenient for this, as the names of labels last unchanged throughout the transformation process. (cf. the names of program variables, which are subject to change due to α-conversion.) Ordinary (unlabeled) tuples may be used instead, but records can coexist more smoothly with our transformation process, which includes flattening and concatenation operations on tuples.

The explanation of the functions given in Fig. 1 follows below. The parsing function `parse` takes one accumulating parameter labeled by `se` and returns two results labeled by `sp` and `l`, where the `se` field holds a stack that remembers the number of parentheses yet to match and the `sp` field represents the parsing stack. The result of parsing is returned in the field `l`. When a malformed input sequence is fed to the `parse` function with `empty` being the initial value of the `se` field, an error will be signaled due to a "pop" operation on an empty stack.

The `cascade` function takes one accumulating parameter labeled by `h` and returns one result labeled by `l`. Given `Leaf` as the initial accumulating parameter, `cascade` recursively replaces the leftmost leaf node in every binary tree with the right subtree of the root node of that tree. This results in a binary tree where all the input binary nodes are reordered onto the spine of left branches in the output. For instance, a binary tree corresponding to the sequence ()(()()) is transformed into the one corresponding to (((()))).

The `unparse` function translates back the binary tree representation into the sequence representation. It takes one accumulating parameter labeled by `k`, whose initial value is `Nil`, and returns one result labeled by `j`.

In the following sections, we will derive a program that fuses these three functions. The result of fusion is given in Fig. 5 of Section 4. The resulting program does not produce any intermediate binary tree representation; it is defined as a first-order function that recurses over the input sequence with a new set of accumulating parameters and return results, which may circularly refers to each other.

3. FUSION WITH ACCUMULATING PARAMETERS

In this section we demonstrate our transformation algorithm by applying it for fusing the producer function `cascade` and the consumer function `unparse`.

Fig. 2 illustrates the outline of the transformation process.[2] Our transformation algorithm is *type-directed*, in a sense that each transformation step corresponds to a type conversion. (In Fig. 2, types that are subject to conversion are underlined.) It first applies shortcut fusion [8] for unifying the producer function `prod` and the consumer function `cons`. The result is a higher-order function `prod'`, which is turned into a first-order function in the remainder of transformation steps. This section demonstrates a simplified version of our transformation algorithm and does not deal with transformation steps (II) and (IV), which concern stack types. We will discuss the general case, allowing stacks and multiple accumulating parameters and return results, in the following section.

[2]For the sake of readability, the figure shows the case where each function takes one accumulating parameter and returns one result, and stacks are only in the accumulating parameter of the producer function.

We will argue the correctness of each transformation step up to observational equivalence. Let us write $M_1 = M_2$ to mean observational equivalence, *i.e.*, M_1 and M_2 show the same behavior under any evaluation context. Under a call-by-name evaluation semantics, $\beta\eta$-equality induces an observational congruence.[3] As for records, the following relation holds.

$$\#l\ (l = M, \ldots) = M \qquad \text{(Rec-Cancel)}$$

We will freely use $\beta\eta$-equality and also (Rec-Cancel) for simplifying programs.

In addition, assuming M is a non-diverging record expression, we will use the following to reason about the correctness of transformation. (In this paper, the type annotation syntax $(M :: \tau)$ represents a typing constraint that M has type τ.)

$$(M\ :: (l_1 :: \tau_1, \ldots, l_n :: \tau_n)) = (l_1 = \#l_1\ M, \ldots, l_n = \#l_n\ M) \qquad \text{(Rec-Eta)}$$

This relation can be safely applied for reasoning about correctness anywhere in this paper, since we assume a particular syntactic form (discussed in Section 5), in which every record is created by the explicit record syntax $(l_1 = M_1, \ldots, l_n = M_n)$ that never diverge.

In this paper, we carry out the discussion on the correctness in a rather conventional way, in order not to include too much theoretical details. (The argument on the correctness in this paper should be formalized in the framework of Pitts' syntactic logical relations [19].) Suppose that we have a program transformation $\mathcal{P}$, whose corresponding type conversion is $\mathcal{T}$. To verify the correctness of transformation, we check if $\mathcal{P}[\![M_1]\!] = \mathcal{P}[\![M_2]\!]$ for every observationally equivalent pair of expressions M_1 and M_2, and the interesting proof case is that for expressions of type τ such that the outermost type constructors in τ and $\mathcal{T}[\![\tau]\!]$ differ. In that case, we show that, for any constructor C and destructor D_i for the type τ, observational equivalence $D_i\ (C\ M_1 \cdots M_n) = M_i$ is preserved by the transformation, *i.e.*, $\mathcal{P}[\![D_i\ (C\ M_1\ \cdots M_n)]\!] = \mathcal{P}[\![M_i]\!]$ holds. As for a record type, this corresponds to verifying that (Rec-Cancel) is preserved by the transformation.

3.1 Shortcut fusion

We apply shortcut fusion to fuse the following function application.

```
unpcas :: T → L
unpcas x =
  #j (unparse (#l (cascade x (h=Leaf))) (k=Nil))
```

[3]We do not consider Haskell's *seq* operator, which breaks this property.

$$
\begin{array}{ll}
\texttt{prod} :: \sigma_1 \to (\texttt{h} :: \texttt{Stk}\,\sigma_2) \to (\texttt{l} :: \sigma_2) \quad \text{and} \quad \texttt{cons} :: \sigma_2 \to (\texttt{k} :: \sigma_3) \to (\texttt{j} :: \sigma_3) & \\
\Downarrow & \text{(I) } \textit{shortcut fusion} \\
\texttt{prod'} :: \sigma_1 \to (\texttt{h} :: \underline{\texttt{Stk}\,((\texttt{k} :: \sigma_3) \to (\texttt{j} :: \sigma_3))}) \to (\texttt{l} :: (\texttt{k} :: \sigma_3) \to (\texttt{j} :: \sigma_3)) & \\
\Downarrow & \text{(II) } \textit{function type lifting for stacks} \\
\texttt{prod'} :: \sigma_1 \to \underline{(\texttt{h} :: \texttt{Stk}\,(\texttt{k} :: \sigma_3) \to \texttt{Stk}\,(\texttt{j} :: \sigma_3))} \to \underline{(\texttt{l} :: (\texttt{k} :: \sigma_3) \to (\texttt{j} :: \sigma_3))} & \\
\Downarrow & \text{(III) } \textit{function type lifting for records} \\
\texttt{prod'} :: \sigma_1 \to ((\texttt{h} :: \underline{\texttt{Stk}\,(\texttt{k} :: \sigma_3)}) \to (\texttt{h} :: \underline{\texttt{Stk}\,(\texttt{j} :: \sigma_3)})) \to ((\texttt{l} :: (\texttt{k} :: \sigma_3)) \to (\texttt{l} :: (\texttt{j} :: \sigma_3))) & \\
\Downarrow & \text{(IV) } \textit{record type lifting} \\
\texttt{prod'} :: \sigma_1 \to (\underline{(\texttt{h} :: (\texttt{k} :: \texttt{Stk}\,\sigma_3))} \to \underline{(\texttt{h} :: (\texttt{j} :: \texttt{Stk}\,\sigma_3))}) \to (\underline{(\texttt{l} :: (\texttt{k} :: \sigma_3))} \to \underline{(\texttt{l} :: (\texttt{j} :: \sigma_3))}) & \\
\Downarrow & \text{(V) } \textit{record type flattening} \\
\texttt{prod'} :: \sigma_1 \to \underline{((\texttt{hk} :: \texttt{Stk}\,\sigma_3) \to (\texttt{hj} :: \texttt{Stk}\,\sigma_3)) \to ((\texttt{lk} :: \sigma_3) \to (\texttt{lj} :: \sigma_3))} & \\
\Downarrow & \text{(VI) } \textit{higher-order removal} \\
\texttt{prod'} :: \sigma_1 \to (\texttt{hj} :: \texttt{Stk}\,\sigma_3, \texttt{lk} :: \sigma_3) \to (\texttt{hk::Stk}\,\sigma_3, \texttt{lj} :: \sigma_3) &
\end{array}
$$

Figure 2: Transformation steps

For this, we rewrite `cascade` and `unparse` in terms of `build` and `fold`, whose definitions on the datatype `T` of binary trees are given below.

```
build :: (∀b. (b → b → b) → b → b) → T
build g = g Node Leaf

fold :: (a → a → a) → a → T → a
fold n l (Node t1 t2) = n (fold n l t1) (fold n l t2)
fold n l Leaf = l
```

The producer function is expressed as `build` g, where g is a function that abstracts over all occurrences of constructors for intermediate data (namely, `Node` and `Leaf`) and `build` reinstantiates the abstractions with concrete constructors. The consumer function is expressed as `fold` $c\ z$, using the generic recursion operator `fold` that replaces every `Node` constructor with c and `Leaf` with z.

The composition of the producer and consumer functions are fused by the following `fold`/`build` rule [8].

$$\texttt{fold}\; c\; z\; (\texttt{build}\; g) \;=\; g\; c\; z$$

Intuitively, this equation means that, instead of applying `fold` $c\ z$ to the intermediate result produced by `build` g, we can obtain the same effect by instantiating abstracted constructors in g with c and z.

The correctness of shortcut fusion follows from free theorems [26, 12], the parametericity property induced from the polymorphic typing constraint on `build`.

The shortcut fusion applies to the present example as follows. (For general methods for translating normal recursive programs into `fold` and `build` forms, see [14, 5].) The consumer function `unparse` is defined with `fold` as follows.

```
unparse :: T → (k:: L) → (j:: L)
unparse = fold c z
  where c t1 t2 =
          \u0 -> let u1 = t1 (k= End (#j u2))
                     u2 = t2 (k= #k u0)
                 in (j= Begin (#j u1))
        z = \u0 -> (j= #k u0)
```

The producer function, as in the form used in the definition of the `unpcas` function above, is expressed as follows:

```
  #1 (cascade x (h=Leaf))
= #1 (build x
       (\n l->cascade' t n l (h=fold n l Leaf)))
```

where `cascade'` is defined as follows.

```
cascade' n l (Node t1 t2) = \x0 ->
  let x1 = cascade' n l t1 (h= n (#l x2, l))
      x2 = cascade' n l t2 (h= #h x0)
  in (l= #l x1)
cascade' n l Leaf = \x0 -> (l= #h x0)
```

We note that we need to abstract over constructors not only in the function body of `cascade` but also its accumulating parameters by applying `fold n l`, so that `cascade'` meets the above mentioned polymorphic typing constraint [7].

Application of the `fold`/`build` rule, followed by some simplifications, gives the following result.

```
cascade' :: T → (h:: (k:: L) → (j:: L))
             → (l:: (k:: L) → (j:: L))
cascade' (Node t1 t2) = \x0 ->
  let x1 = cascade' t1
          (h= \u -> (j=Begin (#j
                       ((#l x2) (k=End (#k u))))))
      x2 = cascade' t2 (h= #h x0)
  in (l= #l x1)
cascade' Leaf = \x0 -> (l= #h x0)

unpcas :: T → L
unpcas x =
  #j ((#l (cascade' x (h= \x0 -> (j= #k x0))))
      (k=Nil))
```

3.2 Function type lifting

The transformation step that follows shortcut fusion is called *function type lifting*, which corresponds to the following type conversion

from	$(p :: \tau_1 \to \tau_2)$
into	$(p :: \tau_1) \to (p :: \tau_2)$,

where p is either `h` or `l`.

This type conversion is realized by applying a transformation function called $\mathcal{L}$ to every defining equation in the program. $\mathcal{L}$ traverses the program structure and applies the following transformation rules wherever possible.

$$\mathcal{L}[\![(p = (M :: \tau_1 \to \tau_2))]\!] \;=\; \backslash y \texttt{ -> } (p = \mathcal{L}[\![M]\!]\,(\#p\, y))$$
$$\mathcal{L}[\![\#p\,(M :: (p :: \tau_1 \to \tau_2))]\!] = \backslash y \texttt{ -> } \#p\,(\mathcal{L}[\![M]\!]\,(p = y))$$

(In both rules, y denotes a fresh variable.)

We omitted trivial transformation rules that are needed just for traversing the program structure. The same convention applies to the rest of the paper.

According to the discussion in the beginning of this section, we can verify the correctness of the present transformation by checking (Rec-Cancel) is preserved by the transformation.

$$\begin{aligned}&\mathcal{L}[\![\#p\,(p = M)]\!]\\ =&\backslash y \texttt{ -> } \#p\,((\backslash y \texttt{ -> } (p = \mathcal{L}[\![M]\!]\,(\#p\, y)))\,(p = y))\\ =&\backslash y \texttt{ -> } \mathcal{L}[\![M]\!]y \;=\; \mathcal{L}[\![M]\!] \qquad\qquad \text{by } \eta\text{-equality}\end{aligned}$$

The result of function type lifting transformation is given as follows.

```
cascade' :: T → (((h:: (k:: L)) → (h:: (j:: L)))
                → ((l:: (k:: L)) → (l:: (j:: L))))
cascade' (Node t1 t2) =  \x0 ->
  let x1 = cascade' t1
             (\y1 -> (h=(j=Begin (#j (#l (x2
                         (l=(k=End (#k (#h y1))))))))))
      x2 = cascade' t2
             (\y2 -> (h= #h (x0 (h= #h y2))))
  in \y0 -> (l= #l (x1 (l= #l y0)))

cascade' Leaf = \x0 ->
  \y0 -> (l= #h (x0 (h= #l y0)))

unpcas :: T → L
unpcas x =
  #j (#l (cascade' x (\y -> (h=(j= #k (#h y))))
                        (l=(k=Nil))))
```

3.3 Record type flattening

This transformation step intends a type conversion

from	$(p :: (q :: \mathtt{L}))$
into	$(pq :: \mathtt{L})$,

where p is either `h` or `l`, q is either `k` or `j`, and pq is a new label obtained by concatenating the two.

This type conversion represents a transformation that merges any pair of nested record labels p and q into a single one pq, *i.e.*, $(p = (q = M))$ into $(pq = M)$ and $\#q\,(\#p\, M)$ into $\#pq\, M$. Formal transformation rules are given below.

$$\mathcal{F}[\![(p = (M :: (q :: \mathtt{L})))]\!] = (pq = \#q\,\mathcal{F}[\![M]\!])$$
$$\mathcal{F}[\![\#p\,(M :: (p :: (q :: \mathtt{L})))]\!] = (q = \#pq\,\mathcal{F}[\![M]\!])$$

We can check this transformation preserves (Rec-Cancel).

$$\begin{aligned}&\mathcal{F}[\![\#p\,(p = (M :: (q :: \mathtt{L})))]\!]\\ =&(q = \#pq\,(pq = \#q\,\mathcal{F}[\![M]\!]))\\ =&(q = \#q\,\mathcal{F}[\![M]\!]) \;=\; \mathcal{F}[\![M]\!] \qquad \text{by (Rec-Eta)}\end{aligned}$$

The result of record type flattening follows below.

```
cascade' :: T → (((hk:: L) → (hj:: L))
             → ((lk:: L) → (lj:: L)))
cascade' (Node t1 t2) = \x0 ->
  let x1 = cascade' t1
             (\y1 -> (hj=Begin (#lj (x2
                        (lk=End (#hk y1))))))
      x2 = cascade' t2
             (\y2 -> (hj= #hj (x0 (hk= #hk y2))))
  in \y0 -> (lj= #lj (x1 (lk= #lk y0)))

cascade' Leaf = \x0 ->
  \y0 -> (lj= #hj (x0 (hk= #lk y0)))

unpcas :: T → L
unpcas x =
  #lj (cascade' x (\y -> (hj= #hk y)) (lk=Nil))
```

3.4 Higher-order removal

Now we are ready for removing higher-order functions. At the type level, the higher-order removal transformation conducts a type conversion

from	$((\mathtt{hk} :: \mathtt{L}) \to (\mathtt{hj} :: \mathtt{L})) \to ((\mathtt{lk} :: \mathtt{L}) \to (\mathtt{lj} :: \mathtt{L}))$
into	$(\mathtt{hj} :: \mathtt{L}, \mathtt{lk} :: \mathtt{L}) \to (\mathtt{hk} :: \mathtt{L}, \mathtt{lj} :: \mathtt{L})$.

We remark that this type conversion preserves polarity of record fields. That is, record fields that occur positively in the higher-order function type (namely, `hk` and `lj` fields) also appear in positive positions of the resulting first-order function type; so do record fields in negative positions (namely, `hj` and `lk` fields).

In a previous work [17], the author has proposed a transformation algorithm that applies to a class of higher-order functions whose each defining equation has the following specific syntactic form

```
f :: T → ((hk::L)→(hj::L)) → ((lk::L)→(lj::L))
  f (C t1 ... tc) = \x0 ->
    let x1 = f tj(1) (\y1 -> (hj=N1))
                ...                                   (3.1)
        xr = f tj(r) (\yr -> (hj=Nr))
    in \y0 -> (lj=N0)
```

where $r, c \geq 0$, $1 \leq j(i) \leq c$ for every i $(1 \leq i \leq r)$, N_i's belong to a set of first-order expressions defined by the following grammar

$$\begin{aligned}\mathcal{M} \;::=\; &\#\mathtt{hj}\,(x_0\,(\mathtt{hk} = \mathcal{M})) \mid \#\mathtt{lj}\,(x_i\,(\mathtt{lk} = \mathcal{M}))\\ &\mid \#\mathtt{lk}\, y_0 \mid \#\mathtt{hk}\, y_i \mid C\,\mathcal{M}\,\cdots\,\mathcal{M} \quad (1 \leq i \leq r),\end{aligned}$$

and furthermore the uses of variables $x_0, \ldots, x_r$ are *linear*. It follows from the linearity that, for every i and appropriate record labels p and q, there is one and only one expression M such that M occurs in the form $\#p\,(x_i\,(q = M))$. Let us write M_{x_0} for the unique expression M in the form $\#\mathtt{hj}\,(x_0\,(\mathtt{hk} = M))$ and M_{x_i} for the unique expression M in the form $\#\mathtt{lj}\,(x_i\,(\mathtt{lk} = M))$ $(1 \leq i \leq r)$.

Each defining equation of the intermediate transformation result in the previous subsection has the above syntactic form. This holds for any program that is derived from a source program that satisfies certain syntactic condition (involving linearity on the uses of variables), which we will discuss in Section 5.

In what follows, let us explain our higher-order removal transformation method in a rather informal way, since a formal justification requires an intricate discussion on the observational equivalence of programs, which is out of the scope of the present paper. Interested readers are deferred to [17].

In the light of the intended conversion on type, we can guess that the transformation would result in a set of defining equations of the following form.

```
f' :: T → (hj :: L, lk :: L) → (hk :: L, lj :: L)
f' (C t_1 ... t_c) x_0 =
  let x_1 = f' t_{j(1)} (hj=M_1^{hj}, lk=M_1^{lk})
          ...
      x_r = f' t_{j(r)} (hj=M_r^{hj}, lk=M_r^{lk})
  in (lj=M_0^{lj}, hk=M_0^{hk})
```

We need to find appropriate expressions to fill in $M_0^{\mathtt{lj}}$, $M_0^{\mathtt{hk}}$, $M_i^{\mathtt{hj}}$'s, and $M_i^{\mathtt{lk}}$'s in the above program. First, we can observe that expression N_i $(1 \le i \le r)$ in (3.1) is each appropriate for $M_i^{\mathtt{hj}}$, as it represents the one and only one `hj` value computed by the recursive call $f\, t_{j(i)} \cdots$. Similarly, N_0 is appropriate for $M_0^{\mathtt{lj}}$. However, we notice that any occurrence of y_i's in these expressions becomes void in the target program, since there are no binding constructs for them. To fix this ill-definedness, we rewrite every subexpression `#lk`y_0 (`#hk` y_i, *resp.*) into `#lk` x_0 (`#hk` x_i, *resp.*), expecting that bindings for $x_0, \ldots, x_r$ in the target program would give the appropriate `lk` (`hk`, *resp.*) value. We note that this rewrite gives rise to a circular definition, where some record fields returned by a function application are referenced from the argument positions of that function call itself. (This *circular tupling technique* has been demonstrated by Bird [1] and has also been exploited in [24] to deforest in accumulating parameters.)

For the above circular binding to be valid, each `#hk` y_i $(1 \le i \le r)$ in the source program (function f) should denote the same value as `#hk` x_i does in the target program (function f'). Since the function call $f\ t_{j(i)}$ (`\`y_i`->`$\cdots$) in f binds the variable x_0 to the function `\`y_i`->`$\cdots$ and uses it in the form `#hj (`x_0 `(hk=`M_{x_0}`))`, y_i is bound to `(hk=`M_{x_0}`)` eventually. On the other hand, in f', x_i is bound to the result of function call $f'\ t_{j(i)}\ \cdots$, which returns the record `(hk=`$M_0^{\mathtt{hk}}$`,`$\cdots$`)`. Hence, $M_0^{\mathtt{hk}}$ must be M_{x_0}.

We also require that `#lk`y_0 in f denotes the same value as `#lk`x_0 in f'. (The initial call sites of both functions are also assumed to meet this condition, as indicated by Eq. (3.3) that we will see later in this section.) It follows from this requirement that, by a similar discussion as above, $M_i^{\mathtt{lk}}$ must be M_{x_i} for each i $(1 \le i \le r)$.

To complete the transformation, we rewrite `#hj` $(x_0\,(\mathtt{hk} = M))$ into `#hj` x_0, where the `hj` value returned by an application of x_0 in the source program is given in the target program through the `hj` field in the record bound by x_0. Similarly, we rewrite `#lj` $(x_i\,(\mathtt{lk} = M))$ into `#lj` x_i for every i $(1 \le i \le r)$.

By the discussion so far, we transform the function definition in the form (3.1) into the following first-order program

```
f' :: T → (hj :: L, lk :: L) → (hk :: L, lj :: L)
f' (C t_1 ... t_c) x_0 =
  let x_1 = f' t_{j(1)} (hj=H[[N_1]], lk=H[[M_{x_1}]])
          ...
      x_r = f' t_{j(r)} (hj=H[[N_r]], lk=H[[M_{x_r}]])
  in (lj=H[[N_0]], hk=H[[M_{x_0}]])
```

```
cascade' :: T → (hj::L, lk::L) → (hk::L, lj::L)
cascade' (Node t1 t2) x0 =
  let x1 = cascade' t1
             (lk= #lk x0, hj= Begin (#lj x2))
      x2 = cascade' t2
             (lk= End (#hk x1), hj= #hj x0)
  in  (hk= #hk x2, lj= #lj x1)

cascade' Leaf x0 = (hk= #lk x0, lj= #hj x0)

unpcas :: T → L
unpcas x = let r = cascade' x (hj= #hk r, lk=Nil)
           in #lj r
```

Figure 3: Fusion result on `unpcas`

where $\mathcal{H}$ is the transformation function defined by the following set of rules.

$$\mathcal{H}[\![x_i\ M]\!] = x_i \qquad (0 \le i \le r) \qquad (3.2a)$$
$$\mathcal{H}[\![y_i]\!] = x_i \qquad (0 \le i \le r) \qquad (3.2b)$$

The following equation, taken from [17], characterizes the relation between the higher-order function f and the resulting first-order function f':

$$\begin{aligned}&\#\mathtt{lj}\ (f\ L\ G\ (\mathtt{lk} = L'))\\ &= \#\mathtt{lj}\ (f'\ L\ (\mathtt{hj} = \#\mathtt{hj}\ (G\\ &\qquad (\mathtt{hk} = \#\mathtt{hk}\ (f'\ L\ (\mathtt{hj} = \Omega, \mathtt{lk} = L')))),\\ &\qquad \mathtt{lk} = L')),\end{aligned}$$

where L is an expression of type `T`, L' is of type `L`, and G is a function of type $(\mathtt{hk} :: \mathtt{L}) \to (\mathtt{hj} :: \mathtt{L})$.

The two different calls to f' in the RHS of the equation can be unified into a single call with a circular definition, as follows.

$$\begin{aligned}&\#\mathtt{lj}\ (f\ L\ G\ (\mathtt{lk} = L'))\\ &= \mathtt{let}\ r = f'\ L\ (\mathtt{hj} = \#\mathtt{hj}\ (G\ (\mathtt{hk} = \#\mathtt{hk}\ r)), \mathtt{lk} = L')\\ &\quad \mathtt{in}\ \#\mathtt{lj}\ r \end{aligned} \qquad (3.3)$$

It is easy to see the former equation follows from the latter, by unwinding the circular let and exchanging Ω with an appropriate expression. (Remember that Ω is a special symbol that represents arbitrary expression.)

Applying the higher-order removal transformation to `cascade'` and rewriting the definition of `unpcas` according to the equation (3.3), we obtain the fusion result given in Fig. 3.

4. FUSION WITH STACKS

In this section, we carry out the remaining half of transformation. That is, we apply our fusion algorithm to the following program

```
unpcaspar :: L → L
unpcaspar x = unpcas (#1 (parse x (se=empty)))
```

which expands to

```
unpcaspar x =
  let r = cascade' (#1 (parse x (se=empty)))
                   (hj= #hk r, lk=Nil)
  in #lj r
```

This section applies our full transformation algorithm, which refines and augments the one given in the previous section in two points. First, two new transformation steps concerning stack types ((II) and (IV) in Fig. 2) are added. Second, the transformation rules are generalized so that they allow multiple accumulating parameters and return values.

For this generalization, we extend our language with *first-class overloaded functions*, which are expressed by the following syntax.

$$\{\texttt{\textbackslash}(x_1 :: \tau_1)\ \texttt{->}\ M_1\texttt{;}\ \cdots\texttt{;}\ \texttt{\textbackslash}(x_n :: \tau_n)\ \texttt{->}\ M_n\} \quad (n \geq 1)$$

An overloaded function is a collection of functions, from which one that has a matching argument type is selected and evaluated per each application of that overloaded function. (A normal λ-abstraction can be recognized as a special instance of an overloaded function.) The type of an overloaded function is written as a collection of function types

$$(\tau_1 \to \tau_1'; \ldots; \tau_n \to \tau_n') \quad (\text{or } \textstyle\sum_{i=1}^n \tau_i \to \tau_i' \text{ for short}),$$

where $\tau_1', \ldots, \tau_n'$ can be different types. In this paper, every different function that is a member of an overloaded function has a different argument type $(p :: \tau)$ distinguished by the record label p. Therefore no ambiguity arises in the selection of function and the return type of each application of an overloaded function is statically determined.

In the rest of this section, let us write $\vec{p}$ for a non-empty set of record labels $\{p_1, \ldots, p_n\}$ $(n \geq 1)$. We also write $(\vec{p} :: \tau)$ to abbreviate $(p_1 :: \tau, \ldots, p_n :: \tau)$ and also write $\widetilde{\tau}$ to denote either τ or $\texttt{Stk}\ \tau$.

We write σ_1, σ_2, and σ_3 for the types of the input, intermediate, and output data structures, respectively. We also write $\vec{h}$, $\vec{l}$, $\vec{k}$, $\vec{j}$ for disjoint sets of record lables, where $\vec{h}$ are used for labeling accumulating parameters of the producer function and $\vec{l}$ for return results; $\vec{k}$ are used for labeling accumulating parameters of the consumer function and $\vec{j}$ for return results.

In this section, we apply the transformation algorithm to the pair of the producer function `parse` and the consumer function `cascade'`, where $\sigma_1 = \texttt{L}$, $\sigma_2 = \texttt{T}$, $\sigma_3 = \texttt{L}$, $\vec{h} = \{\texttt{se}\}$, $\vec{l} = \{\texttt{l}, \texttt{sp}\}$, $\vec{k} = \{\texttt{hj}, \texttt{lk}\}$, and $\vec{j} = \{\texttt{hk}, \texttt{lj}\}$. We omit some of the intermediate transformation results, in order not to clutter pages with lengthy codes.

4.1 Shortcut fusion

As we did in the previous section, we first apply shortcut fusion. The result is displayed below.

```
parse' ::
   L → (se:: Stk((hj::L,lk::L)→(hk::L,lj::L)))
     → (sp:: Stk((hj::L,lk::L)→(lj::L,hk::L)),
        l:: (hj::L,lk::L)→(hk::L,lj::L))
parse' (Begin t) = \x0 ->
  let x1 = parse' t
        (se=Stk (\u0 -> (hk= #lk u0, lj= #hj u0))
                (#se x0))
  in (l= \u0 ->
        let u1 = (#l x1) (lk= #lk u0,
                          hj=Begin (#lj u2))
            u2 = (hd (#sp x1))
                   (lk=End (#hk u1), hj= #hj u0)
        in  (hk= #hk u2, lj= #lj u1),
      sp=tl (#sp x1))

parse' (End t) = \x0 ->
  let x1 = parse' t (se=tl (#se x0))
  in (l=hd (#se x0), sp=Stk (#l x1) (#sp x1))

parse' Nil = \x0 ->
  (l= \u0 -> (hk= #lk u0, lj= #hj u0), sp=empty)

unpcaspar :: L → L
unpcaspar x =
  let r = #l (parse' x (se= empty))
             (hj= #hk r, lk=Nil)
  in #lj r
```

This is obtained by a similar derivation in Section 3.1. The only difference is that we must abstract over constructors in the accumulating parameter of `parse` by applying `mapS (fold n l)`, where `n` and `l` are abstraction variables and `mapS` is a map function on stack elements defined by:

```
mapS f (Stk a as) = Stk (f a) (mapS f as)
```

For the sake of brevity, we have replaced every reference to pattern matching variables a and as bound by a local definition `Stk` a as`=`S with `hd`S and `tl`S, respectively, where `hd` and `tl` are functions satisfying the following equations.

$$\texttt{hd}\ (\texttt{Stk}\ A\ S) = A \qquad (\text{Stk-Hd})$$
$$\texttt{tl}\ (\texttt{Stk}\ A\ S) = S \qquad (\text{Stk-Tl})$$

4.2 Function type lifting for stacks

We introduce a new transformation step that represents the following type conversion:

from	$\texttt{Stk}(\tau_1 \to \tau_2)$
into	$\texttt{Stk}\,\tau_1 \to \texttt{Stk}\,\tau_2$.

This conversion makes function arrows exposed outside of the enclosing stack type constructors, so that the transformation result is ready for the next transformation step. This is realized by the following set of transformation rules

$$\mathcal{S}[\![(\texttt{Stk}\,A\,M :: \texttt{Stk}(\tau_1 \to \tau_2))]\!] =$$
$$\texttt{\textbackslash}w\ \texttt{->}\ \texttt{Stk}(\mathcal{S}[\![A]\!](\texttt{hd}\ w))(\mathcal{S}[\![M]\!](\texttt{tl}\ w))$$
$$\mathcal{S}[\![\texttt{hd}\ M]\!] = \texttt{\textbackslash}u\ \texttt{->}\ \texttt{hd}\ (\mathcal{S}[\![M]\!]\,(\texttt{Stk}\ u\ \Omega))$$
$$\mathcal{S}[\![\texttt{tl}\ M]\!] = \texttt{\textbackslash}w\ \texttt{->}\ \texttt{tl}\ (\mathcal{S}[\![M]\!]\,(\texttt{Stk}\ \Omega\ w))$$

where w and u are fresh variables.

We remark that the above embedding into the target type $\texttt{Stk}\,\tau_1 \to \texttt{Stk}\,\tau_2$ is not full: the resulting function always maps a stack into another stack of the same length. Furthermore, the function operates on stacks elementwisely. That is, each element in the output stack is determined only by the element at the same depth in the input stack; the other elements in the input stack are never referenced and are represented by Ω in the rules above.

By a similar discussion in Section 3, the correctness of this transformation is verified by checking if the transformation preserves (Stk-Hd) and (Stk-Tl). We show the former case only; the verification on the latter is similar.

$$\mathcal{S}[\![\texttt{hd}\ (\texttt{Stk}\,A\,M)]\!]$$
$$= \texttt{\textbackslash}w\ \texttt{->}\ \texttt{hd}\ ((\texttt{\textbackslash}w'\ \texttt{->}\ \texttt{Stk}\,(\mathcal{S}[\![A]\!](\texttt{hd}\ w'))\,(\mathcal{S}[\![M]\!](\texttt{tl}\ w')))\ (\texttt{Stk}\,w\,\Omega))$$
$$= \texttt{\textbackslash}w\ \texttt{->}\ \mathcal{S}[\![A]\!]w\ =\ \mathcal{S}[\![A]\!] \qquad \text{by (Stk-Hd) and } \eta\text{-equality}$$

4.3 Function type lifting for records

This transformation step is a generalization of the one given in Section 3.2 and corresponds to the following type conversion:

$$\begin{array}{ll} \text{from} & (p_1 :: \tau_1 \to \tau_1', \ldots, p_n :: \tau_n \to \tau_n') \\ \text{into} & ((p_1 :: \tau_1) \to (p_1 :: \tau_1');\ \cdots\ ; (p_n :: \tau_n) \to (p_n :: \tau_n')) \end{array}$$

where $n \geq 1$ and $\vec{p} = \vec{h}$ or $\vec{p} = \vec{l}$.

This conversion is realized by the following set of transformation rules

$$\begin{array}{l} \mathcal{L}[\![(p_1 = (M_1 :: \tau_1 \to \tau_1'), \ldots, p_n = (M_n :: \tau_n \to \tau_n'))]\!] = \\ \qquad \{\backslash(y :: (p_1 :: \tau_1))\ \texttt{->}\ (p_1 = \mathcal{L}[\![M_1]\!](\#p_1\, y)); \\ \qquad\qquad\qquad \cdots ; \\ \qquad\ \backslash(y :: (p_n :: \tau_n))\ \texttt{->}\ (p_n = \mathcal{L}[\![M_n]\!](\#p_n\, y))\} \\ \mathcal{L}[\![\#p\,(M :: (p_1 :: \tau_1 \to \tau_1, \ldots, p_n :: \tau_n \to \tau_n'))]\!] = \\ \qquad \backslash v\ \texttt{->}\ \#p(\mathcal{L}[\![M]\!](p = v)) \end{array}$$

where y and v are fresh variables.

In this transformation step, the set of functions in a record $R = (p_1 = F_1, \ldots, p_n = F_n)$ is translated into an overloaded function $F = \{F_1'; \cdots ; F_n'\}$, where each function F_i' has type $(p_i :: \tau_i) \to (p_i :: \tau_i')$. An application of the function F_i in the form $(\#p_i\ R)M$ in the source program is interpreted in the target program as $\#p_i\ (F\ (p_i = M))$, where the corresponding function F_i' in the overloaded function is selected up to the record label p_i. A similar translation may be possible by using a function of type $\langle p_1 : \tau_1, \ldots, p_n : \tau_n\rangle \to \langle p_1 : \tau_1', \ldots, p_n : \tau_n'\rangle$ that operates on disjoint sum types.[4] However, this not only adds extra syntactic verbosity but also gives an imprecision in typing: it blurs the fact that the function maps every input with a record (or injection) label into an output with the same label.

The correctness of this transformation is verified by a derivation similar to that in Section 3.2.

4.4 Record type lifting

Here we need another new transformation step that operates on stack types:

$$\begin{array}{ll} \text{from} & \texttt{Stk}(\vec{q} :: \tau) \\ \text{into} & (\vec{q} :: \texttt{Stk}\,\tau), \end{array}$$

where $\vec{q} = \vec{j}$ or $\vec{q} = \vec{k}$.

The following transformation rules realize such type conversion.

$$\begin{array}{l} \mathcal{R}[\![(\texttt{Stk}\ A\ M :: \texttt{Stk}(\vec{q} :: \tau))]\!]\ = \\ \qquad (q_1 = \texttt{Stk}\ (\#q_1\ \mathcal{R}[\![A]\!])\ (\#q_1\ \mathcal{R}[\![M]\!]),\ \ldots, \\ \qquad\ q_m = \texttt{Stk}\ (\#q_m\ \mathcal{R}[\![A]\!])\ (\#q_m\ \mathcal{R}[\![M]\!])) \\ \mathcal{R}[\![\texttt{hd}\ \ (M :: \texttt{Stk}(\vec{q} :: \tau))]\!] = \\ \qquad (q_1 = \texttt{hd}\ \ (\#q_1\ \mathcal{R}[\![M]\!]), \ldots q_m = \texttt{hd}\ \ (\#q_m\ \mathcal{R}[\![M]\!])) \\ \mathcal{R}[\![\texttt{tl}\ \ (M :: \texttt{Stk}(\vec{q} :: \tau))]\!] = \\ \qquad (q_1 = \texttt{tl}\ \ (\#q_1\ \mathcal{R}[\![M]\!]), \ldots q_m = \texttt{tl}\ \ (\#q_m\ \mathcal{R}[\![M]\!])) \end{array}$$

A similar remark in Section 4.2 applies to this transformation. That is, the embedding into the target type is not full in the sense that every field in the resulting record holds a stack of the same length.

The following derivation verifies that the transformation preserves (STK-HD).

$$\begin{array}{ll} \mathcal{R}[\![\texttt{hd}\ \ (\texttt{Stk}\ A\ M :: \texttt{Stk}(\vec{q} :: \tau))]\!] & \\ = (q_1 = \#q_1\ \mathcal{R}[\![A]\!], \ldots, q_m = \#q_m\ \mathcal{R}[\![A]\!]) & \text{by (REC-CANCEL)} \\ = \mathcal{R}[\![A]\!] & \text{by (REC-ETA)} \end{array}$$

The derivation for (STK-TL) is similar.

4.5 Record type flattening

At this transformation step, we flatten nested record types. We generalize the transformation given in Section 3.3 so that it can handle records with multiple fields.

Let us write $p\vec{q}$ for the set of labels $\{pq \mid q \in \vec{q}\}$. The generalized transformation represents a type conversion

$$\begin{array}{ll} \text{from} & (p :: (\vec{q} :: \vec{\tau})) \\ \text{into} & (p\vec{q} :: \vec{\tau}), \end{array}$$

where $p \in \vec{h} \cup \vec{l}$ and either $\vec{q} = \vec{k}$ or $\vec{q} = \vec{j}$.

The transformation rules are given below.

$$\begin{array}{l} \mathcal{F}[\![(p = (M :: (\vec{q} :: \tau))]\!] = \\ \qquad (pq_1 = \#q_1\ \mathcal{F}[\![M]\!], \ldots pq_m = \#q_m\ \mathcal{F}[\![M]\!]) \\ \mathcal{F}[\![\#p\ (M :: (p :: (\vec{q} :: \tau)))]\!] = \\ \qquad (q_1 = \#pq_1\ \mathcal{F}[\![M]\!], \ldots, q_m = \#pq_m\ \mathcal{F}[\![M]\!]) \end{array}$$

The correctness of this transformation is verified by a derivation similar to that in Section 3.3.

The result of this transformation step is given in Fig. 4, where we saved space by writing `let` U `in (`p_1`=`M_1`, ...,` p_n`=`M_n`)` instead of `(`p_1`=let` U `in` M_1`, ...,` p_n`=let` U `in` M_n`)`.

4.6 Higher-order removal

This section presents a higher-order removal transformation method that generalizes the one given in Section 3.4. In the following, let us write $\vec{p}\vec{q}$ for the set of labels $\{pq \mid p \in \vec{p}, q \in \vec{q}\}$. We write $\prod_{p \in \vec{p}}(p = M_p)$ to abbreviate $(p_1 = M_{p_1}, \ldots, p_n = M_{p_n})$ and also write $\prod_{p\in\vec{p}}(p = M_p) \otimes \prod_{q \in \vec{q}}(q = N_q)$ to denote the record expression $(p_1 = M_{p_1}, \ldots, p_n = M_{p_n},\ q_1 = N_{q_1}, \ldots, q_m = N_{q_m})$.

The present transformation can be explained as a type conversion

$$\begin{array}{ll} \text{from} & \sum_{i=1}^{n}(h_i\vec{k} :: \widetilde{\sigma_3}) \to (h_i\vec{j} :: \widetilde{\sigma_3}) \\ & \qquad\qquad \to\ \sum_{i=1}^{m}(l_i\vec{k} :: \widetilde{\sigma_3}) \to (l_i\vec{j} :: \widetilde{\sigma_3}) \\ \text{into} & (\vec{h}\vec{j} :: \widetilde{\sigma_3}, \vec{l}\vec{k} :: \widetilde{\sigma_3})\ \to\ (\vec{h}\vec{k} :: \widetilde{\sigma_3}, \vec{l}\vec{j} :: \widetilde{\sigma_3}). \end{array}$$

The transformation steps so far would result in a higher-order function f, whose each defining equation has the following form.

$$\begin{array}{l} f\ ::\ \sigma_1\ \to\ \sum_{i=1}^{n}(h_i\vec{k} :: \widetilde{\sigma_3}) \to (h_i\vec{j} :: \widetilde{\sigma_3}) \\ \qquad\qquad \to\ \sum_{i=1}^{m}(l_i\vec{k} :: \widetilde{\sigma_3}) \to (l_i\vec{j} :: \widetilde{\sigma_3}) \\ f\ \ (C\,t_1\ \ldots\ t_c)\ \texttt{=}\ \backslash x_0\ \texttt{->}\ \texttt{let}\ x_1\ \texttt{=}\ f\ \ t_{j(1)}\ \ F_1 \qquad (4.1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ \qquad\qquad\qquad\qquad\qquad x_r\ \texttt{=}\ f\ \ t_{j(r)}\ \ F_r \\ \qquad\qquad\qquad\qquad\quad \texttt{in}\ F_0 \end{array}$$

[4] $\langle p_1 : \tau_1, \ldots, p_n : \tau_n\rangle$ represents a disjoint sum type with n distinct injection labels into the sum type.

```
parse' :: L → ((selk::Stk L,sehj::Stk L) → (sehk::Stk L,selj::Stk L))
            → ((lhj::L,llk::L) → (lhk::L,llj::L);  (sphj::Stk L,splk::Stk L) → (splj::Stk L,sphk::Stk L))
parse' (Begin t) = \x0 ->
  let x1 = parse' t (\y1 -> (sehk=Stk (hd (#selk y1)) (#sehk (x0 (selk=tl (#selk y1), sehj=tl (#sehj y1)))),
                             selj=Stk (hd (#sehj y1), (#selj (x0 (selk=tl (#selk y1), sehj=tl (#sehj y1)))))))
  in {\(y0 :: (lhj::L,llk::L)) ->
       let u1 = (hk= #lhk (x1 (llk= #llk y0, lhj=Begin (#lj u2))), lj= #llj (x1 (llk= #llk y0, lhj=Begin (#lj u2))))
           u2 = (hk= hd (#sphk (x1 (splk=Stk (End (#hk u1)) Ω, sphj=Stk (#lhj y0) Ω))),
                 lj= hd (#splj (x1 (splk=Stk (End (#hk u1)) Ω, sphj=Stk (#lhj y0) Ω))))
       in (lhk= #hk u2, llj= #lj u1) ;
     \(y0 :: (sphj::Stk L,splk::Stk L)) ->
        (sphk=tl (#sphk (x1 (splk=Stk Ω (#splk y0), sphj=Stk Ω (#sphj y0)))),
         splj=tl (#splj (x1 (splk=Stk Ω (#splk y0), sphj=Stk Ω (#sphj y0))))) }

parse' Nil = \x0 -> {\(y0 :: (lhj::L,llk::L)) -> (lhk= #llk y0, llj= #lhj y0);
                     \(y0 :: (sphj::Stk L,splk::Stk L)) -> (sphk=empty, splj=empty) }

unpcaspar :: L → L
unpcaspar x = let r = parse' x (\w->(sehk=empty, selj=empty)) (lhj= #lhk r, llk=Nil) in #llj r
```

Figure 4: Result of recrod type flattening (The End case is omitted.)

where $c, r \geq 0$, $1 \leq j(i) \leq c$ for every i $(1 \leq i \leq r)$, and

$$\begin{aligned}
F_0 \;&=\; \{\backslash y_0 :: (l_1\vec{k} :: \widetilde{\sigma_3}) \texttt{ -> } \textstyle\prod_{l_1 j \in l_1\vec{j}} (l_1 j = N_0^{l_1 j}); \\
&\qquad \cdots ; \\
&\qquad \backslash y_0 :: (l_m\vec{k} :: \widetilde{\sigma_3}) \texttt{ -> } \textstyle\prod_{l_m j \in l_m\vec{j}} (l_m j = N_0^{l_m j})\} \\
F_i \;&=\; \{\backslash y_i :: (h_1\vec{k} :: \widetilde{\sigma_3}) \texttt{ -> } \textstyle\prod_{h_1 j} \in h_1\vec{j}(h_1 j = N_i^{h_1 j}); \\
&\qquad \cdots ; \\
&\qquad \backslash y_i :: (h_n\vec{k} :: \widetilde{\sigma_3}) \texttt{ -> } \textstyle\prod_{h_n j} \in h_n\vec{j}(h_n j = N_i^{h_n j})\}
\end{aligned}$$

In the form above, each N_i^{pq} belongs to a class of expressions specified by the following grammar

$$\begin{aligned}
\mathcal{M} \;::=&\; \#hj\ (x_0\ (\vec{h}\vec{k} = \mathcal{M})) \mid \#lj\ (x_i\ (\vec{l}\vec{k} = \mathcal{M})) \\
&\mid \#lk\ y_0 \mid \#hk\ y_i \mid C\ \mathcal{M}\ \cdots\ \mathcal{M} \\
&\mid \texttt{Stk}\ \mathcal{M}\ \mathcal{M} \mid \texttt{hd}\ \mathcal{M} \mid \texttt{tl}\ \mathcal{M} \mid \Omega \\
&\mid \texttt{let}\ u_1 = \mathcal{M}\ \cdots\ u_v = \mathcal{M}\ \texttt{in}\ \mathcal{M}
\end{aligned}$$

where $1 \leq i \leq r$, $h \in \vec{h}$, $k \in \vec{k}$, $l \in \vec{l}$, $j \in \vec{j}$.

The higher-order removal transformation method in this section extends the one given in Section 3.4 so that multiple record fields are properly handled. For this, for each different overloaded function g occurring in f and each different record label p, we need to identify a unique expression M such that $(p = M)$ is the argument to g. In Section 3.4, such identification was possible, resorting to the linearity on a set of variables. However, in the present generalized setting, the linearity does not necessarily holds for the following reasons. First, the rewrite into `hd` and `tl` notations in Section 4.1 and the transformation rules in Section 4.4 and 4.5 may duplicate expressions, thus breaking the linearity. Second, the transformation rules in Section 4.2 apply the same function $\mathcal{S}[\![M]\!]$ to different arguments, namely, $\texttt{Stk}\ u\ \Omega$ and $\texttt{Stk}\ \Omega\ w$. Finally, a function in the target program may not have any record fields for some labels, when the corresponding fields are not referenced at all in the source program.

We remedy this problem by adopting a relaxed linearity condition described below. We allow the same function to be applied in two or more application sites. If a function is applied to different arguments which have an identical expression for the same field, we recognize the identical expression as the unique expression for that field. Suppose otherwise, *i.e.*, different expressions are assigned for the same field. We can observe that such a set of incompatible expressions is produced only by the transformation rules in Section 4.2, which transform the same expression into a pair of expressions where Ω appears in complementary positions. This indicates that we can always make the set S of incompatible expressions turned into an identical expression, written lub S, by replacing the occurrences of Ω with appropriate expressions.

The formal definition of lub S is as follows: let us write $M \preceq N$ if N is obtained by replacing some of the occurrences of Ω in M with arbitrary expressions. Then the binary relation $\preceq$ induces a partial order with Ω being the least element. For any finite set S of expressions, we write lub S to denote the least upper bound of S w.r.t. $\preceq$, if any. In particular, lub $\emptyset = \Omega$.

As an example of the relaxed linearity condition, consider the program given in Fig. 4. In the defining equation of the `Begin` case, `x0` has two application sites, in both of which the `selk` field of the argument is assigned `tl (#selk y1)` and the `sehj` field is assigned `tl (#sehj y1)`. Hence the expressions for the `selk` field and the `sehj` field are uniquely determined. In the same defining equation, `x1` is applied to different arguments that have incompatible expressions for the same field: for example, there are two different expressions for the `splk` field, namely, $M_1 = $ `Stk` Ω `(#splk y0)` and $M_2 = $ `Stk (End (#hk u1))` Ω. These expressions has the least compatible expression lub $\{M_1, M_2\} = $ `Stk (End (#hk u1)) (#splk y0)`, which we use as the unique expression for the `splk` field. Similarly we have `Stk (#lhj y0) (#sphj y0)` for the `sphj` field.

In case there is no corresponding application site of a function for some record field, we have lub $\emptyset = \Omega$ and consider Ω as the unique expression for that field. This means we consider as if there were a virtual argument expression, which is a record whose every field is set Ω. For example, in the defining equation for the case of `Nil` in Fig. 4, there is no application site of `x0`. So we consider as if there were an invisible application site `x0 (selk=`Ω`,sehj=`Ω`)`. We claim this is valid, since these `selk` and `sehj` fields are never referenced in the source program and hence they would represent unreachable points in the target program too.

The formal definition of the relaxed condition follows below.

```
parse' :: L → (lhj::L, llk::L, sehk::Stk L, selj::Stk L, sphj::Stk L, splk::Stk L)
           → (sehj::Stk L, selk::Stk L, llj::L, lhk::L, splj::Stk L, sphk::Stk L)
parse' (Begin t) x0 =
  let x1 = parse' t (sehk= Stk (hd (#selk x1)) (#sehk x0), selj= Stk (hd (#sehj x1)) (#selj x0), llk= #llk x0,
                     lhj= Begin (hd (#splj x1)), splk= Stk (End (#lhk x1)) (#splk x0), sphj= Stk (#lhj x0) (#sphj x0))
  in (lhk= hd (#sphk x1), llj= #llj x1, sphk= tl (#sphk x1),
      splj= tl (#splj x1), selk= tl (#selk x1), sehj= tl (#sehj x1))

parse' (End t) x0 =
  let x1 = parse' t (sehk= tl (#sehk x0), selj= tl (#selj x0), llk= hd (#splk x0),
                     lhj= hd (#sphj x0), splk= tl (#splk x0), sphj= tl (#sphj x0))
  in (lhk= hd (#sehk x0), llj= hd (#selj x0), sphk= Stk (#lhk x1) (#sphk x1),
      splj= Stk (#llj x1) (#splj x1), selk= Stk (#llk x0) (#selk x1), sehj= Stk (#lhj x0) (#sehj x1))

parse' Nil x0 = (lhk= #llk x0, llj= #lhj x0, sphk=empty, splj=empty, selk=Ω, sehj=Ω)

unpcaspar :: L → L
unpcaspar x = let r = parse' x (sehk=empty, selj=empty, lhj= #lhk r, llk=Nil, sphj=Ω, splk=Ω) in #llj r
```

Figure 5: Final transformation result

Definition 1. *A defining equation of the form* (4.1) *is called* linear *if*

$$M_i^{pq} \;=\; \text{lub}\,\{M \mid M \textit{ appears in the form } x_i\,(pq = M, \cdots)\}$$

is well-defined for every i and pq such that $i = 0$ and $pq \in \vec{h}\vec{k}$ or $1 \leq i \leq r$ and $pq \in \vec{l}\vec{k}$.

This relaxed linearity condition is always satisfied when the original source program meets a certain linearity condition, as we will discuss in the following section.

The higher-order removal transformation in Section 3.4 can be generalized (to allow records with multiple field labels, $\vec{h}\vec{j}$, $\vec{l}\vec{k}$, $\vec{h}\vec{k}$, and $\vec{l}\vec{j}$) as follows. Every defining equation (4.1) of the higher-order function f is transformed into the following defining equation of the first-order function f'.

$$\begin{array}{l} f' \;::\; \sigma_1 \;\rightarrow\; (\vec{h}\vec{j} :: \widetilde{\sigma_3}, \vec{l}\vec{k} :: \widetilde{\sigma_3}) \;\rightarrow\; (\vec{h}\vec{k} :: \widetilde{\sigma_3}, \vec{l}\vec{j} :: \widetilde{\sigma_3}) \\ f' \;\; (C\,t_1\,\ldots\,t_c)\;\; x_0 \;\texttt{=}\; \texttt{let}\;\; x_1 \;\texttt{=}\; f'\;\; t_{j(1)}\;\; F_1' \\ \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ \qquad\qquad\qquad\qquad\quad\; x_r \;\texttt{=}\; f'\;\; t_{j(r)}\;\; F_r' \\ \qquad\qquad\qquad\qquad \texttt{in}\;\; F_0' \end{array} \tag{4.2}$$

where F_0' and F_i' $(0 \leq i \leq r)$ are defined by

$$\begin{array}{lcl} F_0' &=& \prod_{lj \in \vec{l}\vec{j}}(lj = \mathcal{H}[\![N_0^{lj}]\!]) \otimes \prod_{hk \in \vec{h}\vec{k}}(hk = \mathcal{H}[\![M_0^{hk}]\!]) \\ F_i' &=& \prod_{hj \in \vec{h}\vec{j}}(hj = \mathcal{H}[\![N_i^{hj}]\!]) \otimes \prod_{lk \in \vec{l}\vec{k}}(lk = \mathcal{H}[\![M_i^{lk}]\!]) \end{array}$$

with $\mathcal{H}$ being the transformation specified by the following set of rules.

$$\begin{array}{l} \mathcal{H}[\![\texttt{let}\;\; u_1 = M_1 \;\cdots\; u_v = M_v \;\; \texttt{in}\;\; M]\!] \\ \qquad = \mathcal{H}[\![M[u_1 \backslash M_1, \ldots, u_v \backslash M_v]]\!] \end{array} \tag{4.3a}$$

$$\mathcal{H}[\![x_i\; M]\!] = x_i \qquad (0 \leq i \leq r) \tag{4.3b}$$

$$\mathcal{H}[\![y_i]\!] = x_i \qquad (0 \leq i \leq r) \tag{4.3c}$$

We note that the transformation rule (4.3a) never causes infinite unwinding, since we can prove that, in every mutually recursive local definition $\texttt{let}\;\; u_1 = M_1 \;\cdots\; u_v = M_v$ $\texttt{in}\;\; M$ (appearing in the intermediate transformation result), every variable u_k $(1 \leq k \leq v)$ occurs in one of the expressions $M_1, \ldots, M_v$ *behind* some x_i, *i.e.*, in the form $x_i\,(\cdots u_k \cdots)$. This, together with the rule (4.3b), prevents nested unwinding.

The equation (3.3) in Section 3.4 is generalized as follows. For every $l_0 \in \vec{l}$ and $j \in \vec{j}$,

$$\begin{array}{l} \quad \#l_0 j\;(f\;L\;G\;(\prod_{lk \in \vec{l}\vec{k}}(lk = L_{lk}'))) \\ = \texttt{let}\;\; r = \; f'\;L\;(\prod_{hj \in \vec{h}\vec{j}} = \#hj\;(G \\ \qquad\qquad (\prod_{hk \in \vec{h}\vec{k}}(hk = \#hk\;r) \otimes \prod_{lk \in \vec{l}\vec{k}}(lk = L_{lk}'')))), \\ \quad \texttt{in}\;\; \#l_0 j\; r \end{array}$$

where $L_{lk}'' = L_{l_0 k}'$ if $l = l_0$; otherwise, $L_{lk}'' = \Omega$.

Applying this higher-order removal transformation to `parse'`, we obtain the final transformation result given in Fig. 5.

5. CONDITION FOR SUCCESSFUL TRANSFORMATION

This section discusses the sufficient condition on the source program (*i.e.*, the pair of a producer and a consumer function) for our transformation algorithm to succeed.

Let us assume that each defining equations of the producer and consumer functions are given in the following form

$$\begin{array}{l} \texttt{f}\;\; (C\,t_1\;\cdots\;t_c)\;\; x_0 \;\texttt{=} \\ \quad \texttt{let}\;\; x_1 \;\texttt{=}\; \texttt{f}\;\; t_{j(1)}\;\; (p_1 = M_1^{p_1}, \ldots, p_n = M_1^{p_n}) \\ \qquad\quad \cdots \\ \qquad\;\; x_r \;\texttt{=}\; \texttt{f}\;\; t_{j(r)}\;\; (p_1 = M_r^{p_1}, \ldots, p_n = M_r^{p_n}) \\ \qquad\;\; \texttt{Stk}\;\; a_1\;\; as_1 \;\texttt{=}\; N_1 \\ \qquad\quad \cdots \\ \qquad\;\; \texttt{Stk}\;\; a_m\;\; as_m \;\texttt{=}\; N_m \\ \quad \texttt{in}\;\; (q_1 = M_0^{q_1}, \ldots, q_{n'} = M_0^{q_{n'}}) \end{array} \tag{5.1}$$

where $c, r, m \geq 0$, $1 \leq j(i) \leq c$ for every i $(1 \leq i \leq r)$. We assume M_i^p's, M_0^q's, and N_k's belong to a subset of well-typed first-order expressions, defined by the following grammar.

$$\begin{array}{rl} \mathcal{M} ::= & \#p\,x_0 \mid \#q\,x_i \mid a_s \mid as_s \mid C\,\mathcal{M}\;\cdots\;\mathcal{M} \\ & \mid \texttt{Stk}\,\mathcal{M}\;\mathcal{M} \mid \textit{empty} \mid \Omega \end{array}$$

where $p \in \vec{p}$, $q \in \vec{q}$, $1 \leq i \leq r$, $1 \leq s \leq m$.

Definition 2 (Linearity condition). *A function is called* stack linear *if its every defining equation in the form* (5.1) *has at most one occurrence of expressions a_i and as_i for each i $(1 \leq i \leq m)$.*

A function is called context linear *if it is stack linear and there is at most one occurrence of $\#p\,x_0$ for each $p \in \vec{p}$.*

A function is called recursion linear *if it is stack linear, $j(1), \ldots, j(r)$ are pairwise distinct, and there is at most one occurrence of $\#q\, x_i$ for each i $(0 \leq i \leq r)$ and $q \in \vec{q}$.*

Intuitively, a context linear function is constrained to have at most one reference to each accumulating parameter (indexed by a distinct record label); a recursion linear function is constrained to visit every substructure of a node of the input datatype at most once and also every field in the returned record is referenced at most once.

We can show, subject to the following condition on the source program, that the linearity condition required for the higher-order transformation in Section 4.6 is satisfied and hence the transformation succeeds.

Theorem 1 (Successful transformation). *The transformation in Section 4 succeeds, if the producer is a context linear function and the consumer is a recursion linear function.*

We can also show that a closure property holds under a stronger condition.

Theorem 2 (Closure). *If the transformation in Section 4 is applied to a pair of a producer function and a consumer function that are both context and recursion linear, then the result is also a context and recursion linear function.*

In the previous sections, we have implicitly exploited this property. The functions we have considered so far are all both context and recursion linear and hence we were able to apply our transformation algorithm repeatedly to derive a single function that fuses three functions.

6. RELATED WORK

To the best of author's knowledge, there has been no proposal of a fusion transformation for functions that allows both stacks and accumulating parameters. Recent studies on fusion has seen significant advances in dealing with accumulating parameters [13, 21, 25], but none of them deal with stacks.

Our transformation algorithm, as for fusion of functions with accumulating parameters, has a competitive transformation power to Voigtländer's *lazy composition* [25] — both of them allow the pair of a context linear producer function and a recursion linear consumer function. Also, they make use of the circular tupling technique. Interestingly, despite of these similarities, they are based on quite different transformation principles. Lazy composition is an algorithmic instance of unfold/fold-technique [3], while ours are driven by a series of type-directed transformations. We leave more precise comparison to future investigation.

Apart from the researches in functional programming, similar program composition techniques have been studied for attribute grammars as well. Ganzinger and Giegerich [6] developed an algorithm, called *descriptional composition*, where accumulating parameters are allowed as so-called *inherited attributes*. Recently, Nakano [15] extended this composition method to allow stacks.

The present work draws some ideas from their work — use of record labels and flattening of nested records. However, we emphasize that the present work provides an alternative, modular presentation of the same transformation, whereas the attribute grammar composition method has a 'big' transformation step that is hardly modularized. The modularity of our transformation algorithm is achieved by representing intermediate transformation results as higher-order programs, which cannot be expressed in attribute grammars. We also note that our algorithm allows non-recursion linear producer functions (Theorem 1), which cannot be expressed within the limited recursion scheme of attribute grammars.

7. CONCLUSION AND FUTURE WORK

We have presented a new program transformation algorithm for fusion that allows both stacks and accumulating parameters. The proposed algorithm is partly comprised of some of the transformation techniques that have been developed in the communities of functional programming [8, 1, 17] and attribute grammars [6, 15], but reformulates and refines them into a series of type-directed transformations, leading to a logically clean account for the transformation process. We have also shown that the transformation algorithm, under a linearity condition, satisfies a closure property, which allows repeated applications of fusion.

Our algorithm eliminates all the intermediate data structures created by the producer function and yields a first-order program that uses accumulating parameters and stacks, on which parts of the final output are held. However, the elimination of intermediate data structures alone does not necessarily imply a better efficiency, due to the following factors that may degrade the overall run-time efficiency. (i) The record type flattening in Section 4.5 duplicates every record field of the producer function by the number of record labels of the consumer function. This indicates that the execution cost of each recursive call is increased by a constant factor. (ii) Tupling/untupling operations on records arise per each recursive call. In other words, records in the resulting program could be seen as another intermediate structure. (iii) The resulting function includes a circular let, whose execution could be costly.

These factors of deterioration would be compensated by a post-optimization on the resulting program, however. (The concern of this paper is on the derivation of the fusion algorithm and the empirical study on the effect of optimizations discussed below is left to future investigation.) Voigtländer [25] proposed a post-processing method for reducing "ballasts" in his fused programs (tupling/untupling, circular let, etc.) Van Groningen [23] proposed a compiler optimization technique for reducing the overhead raised by allocations of tuples. These techniques would be effectively applied to our fusion results, too. Another prospective direction would be application-specific optimizations. In [16, 18], Nakano and the author developed an algorithm that derives memory efficient XML stream transformers from the result of attribute grammar composition. Refining this method so that it applies to a class of programs involving stacks would be an interesting topic for future research.

In this paper, we consider stacks in producer functions only and defined stacks as a datatype that has only one data constructor. These are not a fundamental restriction of the algorithm, however. It is easy to allow stacks in consumer functions, in which case the resulting program would involve "stacks of stacks". We may also refine our algorithm so that stacks are defined as a datatype with an explicit empty stack constructor and with additional stack constructors as well.

(This would be useful, say, for parsing a language that comprises of two or more kinds of braces.) Putting it more general, we may even allow any polynomial datatypes in place of stacks. Nevertheless, to deal with datatypes that consist of two or more constructors generally, we would need to allow case distinction over these constructors. This would complicate the transformation process to a considerable degree, as so does the extension of the attribute grammar composition with conditionals [2]. A complete solution to this issue would require further elaboration and is left to future work.

Acknowledgment

I am grateful to Masahito Hasegawa and Jacques Garrigue for their valuable comments on an earlier draft. I also thank anonymous reviewers, whose comments were very helpful for improving the paper.

8. REFERENCES

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

[2] J. Boyland and S. L. Graham. Composing tree attributions. In *Proc. of the* 21^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 375–388. ACM Press, 1994.

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

[4] W.-N. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.

[5] O. Chitil. Type inference builds a short cut to deforestation. In *Proc. of International Conference on Functional Programming (ICFP'99)*, pages 249–260. ACM Press, 1999.

[6] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19(6) of *SIGPLAN Notices*, pages 157–170, June 1984.

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

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

[9] The Haskell home page. `http://www.haskell.org/`.

[10] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition, 2001.

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

[12] P. Johann. Short cut fusion is correct. *Journal of Functional Programming*, 13(4):797–814, 2003.

[13] K. Kakehi, R. Glück, and Y. Futamura. An extension of shortcut deforestation for accumulative list folding. *IEICE Transactions on Information and Systems*, E85-D(9):1372–1383, 2002.

[14] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *International Conference on Functional Programming Languages and Computer Architecture, (FPCA'95)*, pages 314–323. ACM Press, 1995.

[15] K. Nakano. Composing stack-attributed tree transducers. Technical Report METR–2004–01, Department of Mathematical Informatics, University of Tokyo, Japan, 2004.

[16] K. Nakano and S. Nishimura. Deriving event-based document transformers from tree-based specifications. In *LDTA'2001 Workshop on Language Descriptions, Tools and Applications*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[17] S. Nishimura. Correctness of a higher-order removal transformation through a relational reasoning. In *Programming Language and Systems, First Asian Symposium, APLAS 2003 Proceedings*, volume 2895 of *LNCS*, pages 358–375. Springer Verlag, 2003. A full version is available as a preprint Kyoto-Math 2003-06 from `http://www.math.kyoto-u.ac.jp/~susumu/papers/aplas03-long.ps.gz`.

[18] S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. submitted.

[19] A. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.

[20] T. Sheard and L. Fegaras. A fold for all seasons. In *International Conference on Functional Programming Languages and Computer (FPCA'93)*, pages 233–242. ACM Press, 1993.

[21] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proc. of the 2002 International Conference on Functional Programming*, 2002.

[22] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. of International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 306–313. ACM Press, 1995.

[23] J. H. van Groningen. Optimising recursive functions yielding multiple results in tuples in a lazy functional language. In *Implementation of Functional Languages, 11th International Workshop, IFL'99*, volume 1868 of *Lecture Notes in Computer Science*, pages 59–76. Springer Verlag, 1999.

[24] J. Voigtländer. Using circular programs to deforest in accumulating parameters. In *ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 126–137. ACM Press, 2002.

[25] J. Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17(1), 2004.

[26] P. Wadler. Theorems for free! In *International Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 347–359. Addison-Wesley, 1989.

[27] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.