Acta Informatica 24, 57–77 (1987)

Acta Informatica

Partial Parametrization Eliminates Multiple Traversals of Data Structures

Masato Takeichi

Department of Computer Science, The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu-shi, Tokyo 182, Japan

Summary. The use of higher order functions in functional programming opens up the possibility of defining functions by partial parametrization, and lazy evaluation brings out a new approach in programming methodology. This paper describes a new transformation technique based on partial parametrization and fully lazy evaluation for eliminating multiple traversals of data structures. It uses no particular mechanisms in functional programming, whereas it transforms a wider class of programs into efficient ones than that proposed so far.

1. Introduction

One of the most important features in functional programming is the use of higher order functions as a powerful mechanism of abstraction. Many similar functions can be defined by parametrizing a higher order function that represents a common pattern of computation. The advantage of programming this way is that modularity can be achieved without introducing any new mechanism except for functional abstraction and application. Such a modular style of programming is of increasing importance in writing large-scale programs. However, programs consisting of these functions sometimes turn out to be inefficient to execute. The main source of the inefficiency lies in the way functions are defined in a program. Each function is usually defined independently by instantiating the common pattern with no reference to the other functions. For example, consider a higher order function that represents a traversal algorithm of a certain data structure. This function is supposed to have a parameter for the operation on each element of the data structure. Then we can define various functions that operate on the data structure by instantiating different operations for the parameter, and use them to construct a program. It is true that the program written in this way attains a high degree of modularity, but multiple traversals of the data structure are inevitable if the program contains multiple instances of the higher order traversal function.

Another important feature in functional programming is that programs can be improved by simple transformations. This is due to the principle of referential transparency of functional programs.

In this paper, we illustrate how these features of functional programming bring unexpected gains in efficiency. The purpose of the transformation technique proposed is, among others, directed to refinement of programs written in a modular style into efficient ones; programs that traverse a data structure more than once are transformed into ones that do so only once.

A similar transformation method by Bird [1] deals with programs of a particular form of functional composition. It relies on lazy evaluation and local recursion to build a circular program structure. Our method can be applied to a wider class of programs than Bird's. It is based on partial parametrization and fully lazy evaluation in addition to local recursion assumed by Bird. Partial parametrization is a basic mechanism in modular programming as described above, and full laziness can be reduced to ordinary laziness by a simple transformation of programs. We can thus make many programs into efficient ones without any particular mechanisms.

We introduce the basic idea of our method through a simple example in Sect. 2. The new transformation technique is described in a general setting in Sect. 3. Further example programs are transformed in Sect. 4. More on the transformation rules are formulated in Sect. 5.

2. Partial Parametrization and Fully Lazy Evaluation

Consider a program to find the average of the elements of an integer list x:

$$
\begin{aligned}
&\mathit{average}\ x = \mathit{DIV}\ (\mathit{sum}\ x)\ (\mathit{length}\ x)\\
&\quad \textbf{whererec}\\
&\qquad \mathit{sum}\ x = \mathit{IF}\ (\mathit{NULL}\ x)\ 0\ (\mathit{PLUS}\ (\mathit{HEAD}\ x)\ (\mathit{sum}\ (\mathit{TAIL}\ x)))\\
&\quad \textbf{and}\\
&\qquad \mathit{length}\ x = \mathit{IF}\ (\mathit{NULL}\ x)\ 0\ (\mathit{PLUS}\ 1\ (\mathit{length}\ (\mathit{TAIL}\ x))).
\end{aligned}
$$

We use a ternary function IF for conditional expressions; IF **true** $e_1\ e_2 = e_1$ and IF **false** $e_1\ e_2 = e_2$ hold. The functions $PLUS$ and DIV are binary arithmetic operations that return the sum and the quotient of two integers, respectively. $NULL$ is a predicate that returns **true** when applied to a null list **nil**, and returns **false** otherwise. $HEAD$ and $TAIL$ are selectors for a nonnull list $(PREFIX\ x\ y)$ with $HEAD(PREFIX\ x\ y) = x$, and $TAIL(PREFIX\ x\ y) = y$. Although predicates and selectors for data structures should have been defined more formally, we leave it for the later sections.

The above program *average* traverses the given list x twice; once for computing the *sum* of the elements and once for finding the *length* of the list. One way to make the program efficient is to combine the two functions *sum* and *length* into a function *sum-length*:

$$\mathit{sum\text{-}length}\ x = [\mathit{sum}\ x,\ \mathit{length}\ x]$$

where $[a, b]$ represents a pair of a and b. A recursive definition of *sum-length* can be synthesized by the *unfold-fold* method [3] and the **where**-*abstraction*:

$$\begin{aligned} &\mathit{sum\text{-}length}\ x = \mathit{IF}\ (\mathit{NULL}\ x)\ [0, 0]\ [\mathit{PLUS}(\mathit{HEAD}\ x)\ s, \mathit{PLUS1}\ l] \\ &\quad \textbf{whererec}\ [s, l] = \mathit{sum\text{-}length}\ (\mathit{TAIL}\ x). \end{aligned}$$

Having defined this function, we can rewrite the function *average* as

$$\begin{aligned} &\mathit{average}\ x = \mathit{DIV}\ s\ l \\ &\quad \textbf{where}\ [s, l] = \mathit{sum\text{-}length}\ x \\ &\qquad \textbf{whererec}\ \mathit{sum\text{-}length}\ x = \ldots \end{aligned}$$

This program traverses the list x only once.

The above transformation method can be applied to programs of the form $h(fx)(gx)$, which we may call *S′-composition* after the **S′** combinator of combinator logic[1]. It is, however, largely dependent on the form of functional composition.

Bird [1] extends this idea of the tupling and the unfold-fold methods to develop a technique for transforming programs of the *S-composition* form, i.e., $fx(gx)$, into ones that traverse a data structure x only once. Bird's transformation relies on *lazy evaluation* [4, 5] and *local recursion* to build a circular program structure. We do not deal with such a program here, but explain the basic idea of our new transformation technique based on *partial parametrization* and *fully lazy evaluation* using the example program *average*. In later sections we will apply the technique to Bird's examples.

First of all, it should be observed that both the functions *sum* and *length* traverse the list x precisely in the same manner. Or rather, one can say that these definitions come out from a common pattern of computation. They differ only in the function that operates on the elements of the list. Hence, the uncommon operation having been made a parameter f, the common parts of these functions can be expressed as a higher order function *accum* that accumulates the results of applying the function f to all the elements of the list x:

$$\begin{aligned} &\mathit{accum}\ f\ x = \\ &\quad \mathit{IF}\ (\mathit{NULL}\ x)\ 0\ (\mathit{PLUS}\ (f\ (\mathit{HEAD}\ x))\ (\mathit{accum}\ f\ (\mathit{TAIL}\ x))). \end{aligned}$$

We can redefine the two functions using *accum*:

$$\begin{aligned} &\mathit{sum} = \mathit{accum}\ \mathbf{I} \quad \textbf{where}\ \mathbf{I}\ x = x \\ &\mathit{length} = \mathit{accum}\ (\mathbf{K}\ 1) \quad \textbf{where}\ \mathbf{K}\ x\ y = x. \end{aligned}$$

The use of higher order functions this way is so common that we can find many illustrative examples in the literatures on functional programming, e.g., [2]. In fact, the function *accum* itself can be defined by instantiating another higher order function that represents a more general pattern of computation (see Sect. 4).

---

[1] Turner [11] introduces the combinator **S′** for $\mathbf{S}'hfgx = h(fg)(gx)$ as an extension to the standard **S** combinator for $\mathbf{S}fgx = fx(gx)$. The combinator **S′** is also written as Φ in some books on combinatory logic. We prefer **S′** to Φ because the latter symbol is used differently in this paper

The transformation technique proposed in this paper uses such a higher order function of which parameters are arranged so that the first one is the data structure x to be traversed. We accordingly use the following version of the accumulation function in place of *accum*:

$$accum'\ x\ f =$$
$$\quad IF\ (NULL\ x)\ 0\ (PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f))$$

and we rewrite the functions *sum* and *length* as

$$sum\ x = accum'\ x\ \mathbf{I}$$
$$length\ x = accum'\ x\ (\mathbf{K}\ 1).$$

Then, we can take advantage of an opportunity to make the common term $(accum'\ x)$ be shared by both of the functions. This term is a unary function obtained by parametrizing only the first argument of the binary function *accum'*. As has been done above, the higher order function that is derived from a commonly used one but takes the data structure as its first argument is denoted by the function name with a prime attached.

Using the above definitions, we can rewrite the original program as

$$average\ x = DIV\ (\xi\ \mathbf{I})\ (\xi\ (\mathbf{K}\ 1))$$
$$\quad \textbf{where}\ \xi = accum'\ x$$
$$\quad\quad \textbf{whererec}\ accum'\ x\ f = \ldots$$

Lazy evaluation with a call-by-need mechanism [13], or call-by-delayed-value [12] ensures that the term ξ appearing in both the arguments of *DIV* is evaluated only once.

Therefore, if the list x has been frozen in the function $\xi = (accum'\ x)$ and no more reference to x occurs in evaluation of (ξf) for any function f, the new program *average* traverses the list x only once. If we were allowed to precompute ξ for a particular x, say $[3; 5]$, we obtain ξ in a single traversal of x as

$$\begin{aligned}
\xi &= accum'\ [3; 5] \\
&= \lambda f.\ IF\ (NULL\ [3; 5])\ 0\ (PLUS\ (f\ (HEAD\ [3; 5]))\ (accum'\ (TAIL\ [3; 5])\ f)) \\
&= \lambda f.\ PLUS\ (f\ 3)\ (IF\ (NULL\ [5])\ 0\ (PLUS\ (f\ (HEAD\ [5]))\ (accum'\ (TAIL\ [5])\ f))) \\
&= \lambda f.\ PLUS\ (f\ 3)\ (PLUS\ (f\ 5)\ (IF\ (NULL\ [\,])\ 0\ (\ldots))) \\
&= \lambda f.\ PLUS\ (f\ 3)\ (PLUS\ (f\ 5)\ 0).
\end{aligned}$$

Using this value for ξ, we can compute the average of $[3; 5]$ according to the equation for *average* without traversing x any more. Precomputation is not satisfactory, however. It is costly to calculate $(accum'\ x)$ as above each time the list x is given. Fortunately, the desired effect is achieved by the use of *fully lazy evaluation*; every expression is evaluated at most once after the variables in it have been bound [6, 7]. In our particular case, fully lazy evaluation ensures that any expression containing the parameter x is evaluated at most once. With an algorithm similar to the one by Hughes for finding super-combinators [6], the definition of *accum'* can be rewritten as

$$accum'\ x = \Phi\ (IF\ (NULL\ x))\ (HEAD\ x)\ (accum'\ (TAIL\ x))$$
$$\quad \textbf{where}\ \Phi\ \beta\ \alpha_1\ \alpha_2\ f = \beta\ 0\ (PLUS\ (f\ \alpha_1)\ (\alpha_2\ f))$$

where Φ is a super-combinator, that is, a closed function with no free variables. The arguments of Φ are maximal terms dependent on x (see Sect. 3).

In lazy evaluation, the arguments of Φ are evaluated at most once as needed within the body of Φ. Once the argument has been evaluated, its result is retained in place for subsequent evaluation. We have thus attained full laziness by means of ordinary lazy evaluation with call-by-need semantics.

By way of illustration, we will trace the computational process for *average* [3; 5]. In that, the function *DIV* forces both the operands be evaluated before division is taken, evaluation of the first operand (ξ **I**) proceeds as

$$\begin{aligned}\xi\,\mathbf{I} &= accum'\,[3;5]\,\mathbf{I}\\ &= \Phi\,(IF\,(NULL\,[3;5]))\,(HEAD\,[3;5])\,(accum'\,(TAIL\,[3;5]))\,\mathbf{I}\\ &= (IF\,(NULL\,[3;5]))\,0\,(PLUS\,(\mathbf{I}\,(HEAD\,[3;5]))\,(accum'\,(TAIL\,[3;5])\,\mathbf{I})).\end{aligned}$$

The first term $(IF\,(NULL\,[3;5]))$ is evaluated to become *IF-FALSE*. The function $IF\text{-}FALSE = (IF\ \mathbf{false})$ satisfies $IF\text{-}FALSE\ e_1\,e_2 = e_2$. Similarly, $IF\text{-}TRUE = (IF\ \mathbf{true})$ and $IF\text{-}TRUE\ e_1\,e_2 = e_1$. Once the result has been obtained, it supersedes the argument for later use. That is, ξ becomes

$$\xi = \Phi\ IF\text{-}FALSE\ (HEAD\,[3;5])\,(accum'\,(TAIL\,[3;5]))$$

and the term (ξ **I**) becomes

$$\xi\,\mathbf{I} = PLUS\,(\mathbf{I}\,(HEAD\,[3;5]))\,(accum'\,(TAIL\,[3;5])\,\mathbf{I}).$$

Evaluation now proceeds to the arguments of *PLUS*. The first one is evaluated to be 3 and the second to be

$$accum'\,(TAIL\,[3;5])\,\mathbf{I} = \Phi\,(IF\,(NULL\,t))\,(HEAD\,t)\,(accum'\,(TAIL\,t))\,\mathbf{I}$$

where $t = TAIL\,[3;5]$ has not yet been evaluated. When that expression is evaluated, $(NULL\,t)$ results in **false** and t becomes [5] as a side-effect. Thus we have

$$\xi = \Phi\ IF\text{-}FALSE\ 3\,(\Phi\ IF\text{-}FALSE\,(HEAD\,[5])\,(accum'\,(TAIL\,[5])))$$

and

$$\xi\,\mathbf{I} = PLUS\,(3\,(\Phi\ IF\text{-}FALSE\,(HEAD\,[5])\,(accum'\,(TAIL\,[5])\,\mathbf{I}))).$$

Finally we obtain

$$\begin{aligned}\xi = {}&\Phi\ IF\text{-}FALSE\ 3\\ &(\Phi\ IF\text{-}FALSE\ 5\\ &\quad(\Phi\ IF\text{-}TRUE\,(HEAD\,[\,])\,(accum'\,(TAIL\,[\,]))))\end{aligned}$$

and

$$\xi\,\mathbf{I} = PLUS\ 3\,(PLUS\ 5\ 0) = 8.$$

When the other argument of *DIV*, i.e., (ξ (**K** 1)), comes to be evaluated, the value ξ that has just been obtained is used. Note that the term

$$\Phi\ IF\text{-}TRUE\,(HEAD\,[\,])\,(accum'\,(TAIL\,[\,]))$$

is immediately reduced to 0 with no reference to *HEAD* or *TAIL* if some argument follows it. Since there is no more *NULL*, *HEAD*, or *TAIL* remaining in ξ, we get the value (ξ (**K** 1)) = 2 without any traversal of the list x.

As we have seen from the above example, our new transformation technique consists of the following novel ideas in functional programming:

(1) For the common expressions to share the result of partial parametrization of the higher order function.
(2) For the partially parametrized function to be evaluated recursively in a fully lazy way.

We assume hereafter that our functional language allows one to deal with partially parametrized functions as first class objects, and that evaluation is done lazily with the call-by-need mechanism. We have already demonstrated a technique of transforming a function definition into one that is evaluated in a fully lazy way using the ordinary lazy evaluation mechanism.

3. Transformation Rules

Our transformation technique exemplified in the last section contains several stages. We describe them in a general setting to formulate the rules. Suppose that a naive functional program is given as

$$
\begin{aligned}
&f x = F \\
&\quad \textbf{whererec} \\
&\qquad g_1 x = \lambda \mathbf{v}_1 \cdot G_1 \\
&\quad \textbf{and} \\
&\qquad \cdots \\
&\quad \textbf{and} \\
&\qquad g_k x = \lambda \mathbf{v}_k \cdot G_k
\end{aligned}
$$

where F contains terms $(g_i x)$ and constants. Each of G_i may well contain lambda variables $\mathbf{v}_i$ in addition to x and constants. In general, G_i may contain $(g_j x)$ for $i \neq j$, so the g's can be mutually recursive. We assume, however, in this section that G_i does not contain $(g_j x)$.

[Generalization]

The first thing to do in our transformation is to find a common recursive form for the traversal functions $g_1, \ldots, g_k$. The higher order function h with parameters $x, y_1, \ldots, y_p$ should be defined so that every traversal function concerned is a particular instance of h applied to x, and p actual arguments. Let h be defined recursively as

$$h x y_1 \ldots y_p = H$$

where H contains $h, x, y_1, \ldots, y_p$ and constants. Then we have

$$g_i x = \lambda \mathbf{v}_i \,.\, h x a_1 \ldots a_p \quad \text{for} \quad i = 1, \ldots, k$$

with $a_1, \ldots, a_p$ chosen appropriately.

[Substitution]

The next step is to replace all the terms $(g_i x)$ in F by

$$\lambda \mathbf{v}_i \,.\, \xi a_1 \ldots a_p$$

and to use **where**-abstraction getting

$f x = F'$
  where
    $\xi = h x$
      whererec
        $h x y_1 \ldots y_p = H.$

[Lambda-hoisting]

To realize full laziness by a call-by-need mechanism, transform the above definition for the function h by hoisting maximal free occurrences of expressions in H with respect to $y_1, \ldots, y_p$.

$h x = \Phi b_1 \ldots b_m$
  where
    $\Phi \beta_1 \ldots \beta_m y_1 \ldots y_p = H'$

where b_i's stand for the expressions each of which occurs maximally free in H and contains only x, h and constants.

Among these stages, only the generalization is heuristic. The other two are done entirely mechanically. Generalization can be done by simple inspection for small programs, whereas general rules and algorithms are to be sought for sizable programs.

On the other hand, generalization may be rather straightforward when programs are constructed from a generic traversal function by means of partial parametrization. Constructing programs in such a style is, indeed, one of the advantageous features of functional programming. In our particular case, we have a special interest in traversal functions for certain data structures. As every traversal function necessarily depends on the concrete data structure, defining an abstract data type would be most appropriate. Using an abstract type definition, the structure of data and the traversal functions on it can be encapsulated into a module. The data structure used in the previous section can be written in Standard ML [10] as

abstype *intlist* = **nil** | *PREFIX* **of int** (**int** *list*)
  with
    val *NULL* **nil** = **true** | *NULL*(*PREFIX* _) = **false**
    and *HEAD*(*PREFIX* *a* *x*) = *a*
    and *TAIL*(*PREFIX* *a* *x*) = *x*
    and rec *accum'* *x* = Φ(*IF*(*NULL* *x*))(*HEAD* *x*)(*accum'* (*TAIL* *x*))
      where $\Phi \beta \alpha_1 \alpha_2 f = \beta 0 (PLUS (f \alpha_1)(\alpha_2 f))$
end.

We have used data constructors **nil** and *PREFIX* corresponding to [ ] and [;] in the previous example. It is true that there may be various kinds of traversal functions for a particular data structure, but generalization becomes trivial if all the $(g_i x)$ are expressed as instances of a generic traversal function in common. We will give some typical data structures and traversal functions in later sections.

The *lambda-hoisting* rule is much similar to the rule by Hughes [6, 7] specifying how to convert a lambda expression into super-combinators. It is slightly different, however, in the treatment of recursive definitions. We will give an algorithm for lambda-hoisting adapted to the definition for h specified as above. For simplicity, we assume here that the right hand side expression H is of the applicative form and it does not contain lambda expressions as its constituents. That is, an applicative expression is either

(1) *simple* and is either
  (1.1) a constant
  or
  (1.2) a variable

or

(2) *compound* and is a combination $(e_1\, e_2)$ where both e_1 and e_2 are applicative expressions. Here, we regard fixed functions like *IF*, *PLUS*, **I**, etc., as constants. We first define *free occurrences* of applicative expressions.

An occurence of an expression e in H is defined to be *free* with respect to $y_1, \ldots, y_p$ as follows.

(1) If e is simple and
  (1.1) if e is a constant, e is not free in H.
  or
  (1.2) if e is a variable, and
    (1.2.1) if e is the function variable h or the variable x,
      e is *free* in H with respect to $y_1, \ldots, y_p$,
    or
    (1.2.2) e is not free in H, otherwise.

(2) If e is compound and is a combination $(e_1\, e_2)$, and
  (2.1) if either of e_1 or e_2 is free in H, and
    (2.1.1) the other one is also free in H, or the other one
      is a constant, e is *free* in H with respect to $y_1, \ldots, y_p$
    or
    (2.1.2) e is not free in H, otherwise.
  (2.2) e is not free in H, otherwise.

Free occurrences of expressions that are not part of any larger free occurrence of an expression are called *maximal free occurrences* of expressions in H with respect to $y_1, \ldots, y_p$.

We assume that all the operations usually written in infix form are expressed by corresponding functions, and the conditional expression is also written using the function *IF* with three arguments. Although there is no problem arisen from this definition, it would be desirable to deal with conditionals as a special form. That is, even in the case that $(IF\, e_1\, e_2)$ happens to be a maximal free occurrence from the above definition, we take both the parts $(IF\, e_1)$ and e_2 as maximal free occurrences.

We can now describe the algorithm for lambda-hoisting.

(Step 1) Identify all the maximal free occurrences of expressions $b_1, \ldots, b_m$ in H with respect to $y_1, \ldots, y_p$.

(Step 2) Replace each occurrence of b_i in H with a new variable β_i for $i = 1, \ldots, m$ to obtain H'.

(Step 3) Construct the definition of the form

$$hx = \Phi b_1 \ldots b_m \quad \textbf{where} \quad \Phi \beta_1 \ldots \beta_m y_1 \ldots y_p = H'.$$

If there exist common sub-expressions in $b_1, \ldots, b_m$, the expression $\Phi b_1 \ldots b_m$ should be further transformed using **where**-abstraction so that they are replaced by the common variables bound to those expressions.

It is desirable, though not absolutely necessary, that identical expressions are not duplicated among $b_1, \ldots, b_m$ in (Step 1). For example, consider the case where there are two maximal free occurrences of $(HEAD\ x)$ in H. If we take them to be distinctively, say b_1 and b_2, we cannot have the advantage of call-by-need evaluation at least as it is. After one of them has been evaluated to yield the value of $(HEAD\ x)$, the result cannot be used in evaluation of the other one. In fact, (Step 3) ensures that such situation is properly dealt with for completing lambda-hoisting. The number of parameters is, of course, greater than the case where common expressions are made into one in (Step 1). On the other hand, even if no duplicated b_i have been extracted in (Step 1), (Step 3) is indispensably necessary. This is because there may exist an expression b_i identical to some proper sub-expression of another expression b_j both of which occur maximally free in H.

As mentioned earlier, lambda-hoisting is intended to realize full laziness by ordinary lazy evaluation. The procedure essentially converts an expression into a new form of expression that uses functions with no free variables. In that, it is most important that maximal free occurrences of expressions are identified and they are moved out as arguments of the functions. This is the basic idea of super-combinators by Hughes. It should be noted, however, that the lambda-hoisting rule differs from that of Hughes for recursive definitions. According to [6], the definition of *accum'* in the previous section can be rewritten as

$$\begin{aligned} accum'\ x &= \lambda f.\ IF\ (NULL\ x)\ 0\ (PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f) \\ &= IF\ (NULL\ x)\ 0\ (\lambda f.\ PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f)) \\ &= IF\ (NULL\ x)\ 0\ ((\lambda x.\ \lambda f.\ PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f))\ x) \end{aligned}$$

and we have a super-combinator Ψ such that

$$\begin{aligned} &accum'\ x = IF\ (NULL\ x)\ 0\ (\Psi\ x) \\ &\quad \textbf{whererec}\ \Psi\ x f = PLUS\ (f\ (HEAD\ x))\ (accum'\ (TAIL\ x)\ f). \end{aligned}$$

In case of $x = [3; 5]$, we get

$$\begin{aligned} \xi\ \mathbf{I} &= accum'\ [3; 5]\ \mathbf{I} \\ &= IF\ (NULL\ [3; 5])\ 0\ (\Psi\ [3; 5])\ \mathbf{I} \\ &= \Psi\ [3; 5]\ \mathbf{I} \\ &= PLUS\ (\mathbf{I}\ (HEAD\ [3; 5]))\ (accum'\ (TAIL\ [3; 5])\ \mathbf{I}) \end{aligned}$$

and the computation proceeds as before to yield the value $(\xi\ \mathbf{I}) = 8$. But the value of ξ after evaluation of $(\xi\ \mathbf{I})$ is simply $(\Psi\ [3; 5])$, and no further reduction

has been taken. This means that another traversal of the data structure is needed in evaluation of $(\xi(\mathbf{K}\,1))$.

Yet another transformation rule called *lambda-lifting* is presented in [9]. The idea of identifying maximal free occurrences of expressions is not included in lambda-lifting. If we extrapolate the idea of lambda-lifting and move out maximal free occurrences of expressions instead of variables alone, we have

$$accum'\,x = \Phi'\,(IF\,(NULL\,x))\,(HEAD\,x)\,(TAIL\,x)$$
$$\textbf{whererec}\ \Phi'\,\beta\,\alpha_1\,\alpha_2\,f = \beta\,0\,(PLUS\,(f\,\alpha_1)\,(accum'\,\alpha_2\,f)).$$

This form of definition fails again to achieve our end. The situation is much similar to the case of Hughes' algorithm for super-combinators.

In summary, it is essential in lambda-hoisting that the variable h representing the traversal function and the variable x for the data structure should be taken to be free with respect to other variables as carefully described in the definition of free occurrences of expressions. As a matter of fact, we have made the data structure be the first argument of the traversal function h so that the combination of h and the expression for selecting some part of the data structure x become a single larger free expression.

4. Functions on Some Data Structures

In this section we will illustrate how the transformation rules are applied to programs that traverse particular data structures. We have already presented our transformation technique for a program of the S'-composition form in Sect. 2. We can equally transform programs of the S-composition form using the rules in the previous section. Example programs in this section are taken from [1] for comparing our transformation technique with Bird's.

4.1. *Functions on Lists*

To begin with, consider a definition of a general list type

```
abstype 'a list = nil | PREFIX of 'a('a list)
    with
        val NULL nil = true | NULL(PREFIX _ _) = false
        and HEAD(PREFIX a x) = a
        and TAIL(PREFIX a x) = x
        and rec ...
end.
```

Traversal functions to be defined in this abstract type are, among others, ones that scan a single list of the type *'a list*. We first define these functions in a commonly used form, and then derive the final equations by exchanging the order of parameters. Such functions are from [2]:

$$list\,1\,a\,g\,f\,x = IF\,(NULL\,x)\,a\,(g\,(f\,(HEAD\,x))\,(list\,1\,a\,g\,f\,(TAIL\,x)))$$
$$list\,2\,a\,g\,f\,x = IF\,(NULL\,x)\,a\,(list\,2\,(g\,(f\,(HEAD\,x))\,a)\,g\,f\,(TAIL\,x)).$$

The functions *list*1 and *list*2 are more general than the function *accum* used in Sect. 2. In fact, we have

$$accum = list1\ 0\ PLUS$$
$$accum = list2\ 0\ PLUS.$$

It should be noted, however, that we can derive the recursion equation of *accum* from *list*1 by replacing the term $(list1\,a\,g)$ on the right hand side with *accum* itself, while we cannot do it from *list*2.

These functions can be used to define many functions. For example, commonly used higher order functions *map* and *filter*:

$$map\ f\ x = IF\ (NULL\ x)\ \mathbf{nil}\ (PREFIX\ (f\ (HEAD\ x))\ (map\ f\ (TAIL\ x))$$
$$filter\ p\ x = IF\ (NULL\ x)\ \mathbf{nil}\ (IF\ (p\ u)\ (PREFIX\ u\ v)\ v)$$
$$\mathbf{whererec}\ u = HEAD\ x\ \mathbf{and}\ v = filter\ p\ (TAIL\ x)$$

can be defined using *list*1 as

$$map = list1\ \mathbf{nil}\ PREFIX$$
$$filter\ p = list1\ \mathbf{nil}\ (\lambda u\ v.\ IF\ (p\ u)\ (PREFIX\ u\ v)\ v)\ \mathbf{I}.$$

And the function *reverse* that reverses the given list can be written using *list*2 as

$$reverse = list2\ \mathbf{nil}\ PREFIX\ \mathbf{I}.$$

We need sometimes functions that operate two lists of the same length as in comparing corresponding elements of the lists. Hence, we next define such functions *zip*1 and *zip*2 that scan lists x and y of the same length. These are based on *list*1 and *list*2, respectively.

$$zip1\ a\ g\ f\ x\ y =$$
$$IF\ (NULL\ x)\ a\ (g\ (f\ (HEAD\ x)\ (HEAD\ y))\ (zip1\ a\ g\ f\ (TAIL\ x)\ (TAIL\ y)))$$
$$zip2\ a\ g\ f\ x\ y =$$
$$IF\ (NULL\ x)\ a\ (zip2\ (g\ (f\ (HEAD\ x)\ (TAIL\ y))\ a)\ g\ f\ (TAIL\ x)\ (TAIL\ y)).$$

Here, the function g combines successively the results of f applied to corresponding elements of the lists x and y. The functions *list*1 and *list*2 are defined using *zip*1 and *zip*2 as

$$list1\ a\ g\ f\ x = zip1\ a\ g\ \bar{f}\ x\ DUMMY$$
$$list2\ a\ g\ f\ x = zip2\ a\ g\ \bar{f}\ x\ DUMMY$$

where

$$\bar{f}\ x\ y = f\ x$$

and *DUMMY* is a dummy expression never to be evaluated though its presence is necessary.

We now turn to defining the final equations for these functions to make the proposed transformation technique be applicable. To do so, we have only to promote the parameter x at the front of the parameter list. As noted in Sect. 2,

we write such function names with primes.

$$list1'\,x\,a\,g\,f = IF\,(NULL\,x)\,a\,(g\,(f\,(HEAD\,x))\,(list1'\,(TAIL\,x)\,a\,g\,f))$$
$$list2'\,x\,a\,g\,f = IF\,(NULL\,x)\,a\,(list2'\,(TAIL\,x)\,(g\,(f\,(HEAD\,x))\,a)\,g\,f)$$
$$zip1'\,x\,a\,g\,f\,y =$$
$$\quad IF\,(NULL\,x)\,a\,(g\,(f\,(HEAD\,x)\,(HEAD\,y))\,(zip1'\,(TAIL\,x)\,a\,g\,f\,(TAIL\,y)))$$
$$zip2'\,x\,a\,g\,f\,y =$$
$$\quad IF\,(NULL\,x)\,a\,(zip2'\,(TAIL\,x)\,(g\,(f\,(HEAD\,x)\,(HEAD\,y))\,a)\,g\,f\,(TAIL\,y)).$$

By lambda-hoisting, we get the equations

$$list1'\,x = \Phi\,(IF\,(NULL\,x))\,(HEAD\,x)\,(list1'\,(TAIL\,x))$$
$$\quad \textbf{where}\ \Phi\,\beta\,\alpha\,\eta\,a\,g\,f = \beta\,a\,(g\,(f\,\alpha)\,(\eta\,a\,g\,f))$$
$$list2'\,x = \Phi\,(IF\,(NULL\,x))\,(HEAD\,x)\,(list2'\,(TAIL\,x))$$
$$\quad \textbf{where}\ \Phi\,\beta\,\alpha\,\eta\,a\,g\,f = \beta\,a\,(\eta\,(g\,(f\,\alpha)\,a)\,g\,f)$$
$$zip1'\,x = \Phi\,(IF\,(NULL\,x))\,(HEAD\,x)\,(zip1'\,(TAIL\,x))$$
$$\quad \textbf{where}\ \Phi\,\beta\,\alpha\,\eta\,a\,g\,f\,y = \beta\,a\,(g\,(f\,\alpha\,(HEAD\,y))\,(\eta\,a\,g\,f\,(TAIL\,y)))$$
$$zip2'\,x = \Phi\,(IF\,(NULL\,x))\,(HEAD\,x)\,(zip2'\,(TAIL\,x))$$
$$\quad \textbf{where}\ \Phi\,\beta\,\alpha\,\eta\,a\,g\,f\,y = \beta\,a\,(\eta\,(g\,(f\,\alpha\,(HEAD\,y))\,a)\,g\,f\,(TAIL\,y)).$$

We assume here that these definitions are included in the above abstract type definition of *'a list*.

As an example of list traversal programs, consider the palindrome problem in [1]: Determine whether a given list of integers is palindromic; i.e., equal to its reverse.

A straightforward solution looks like

$$palindrome\,x = eqlist\,x\,(reverse\,x)$$
$$\quad \textbf{whererec}$$
$$\quad\quad eqlist\,x\,y =$$
$$\quad\quad\quad IF\,(NULL\,x)\ \textbf{true}$$
$$\quad\quad\quad\quad (AND\,(EQUAL\,(HEAD\,x)\,(HEAD\,y))\,(eqlist\,(TAIL\,x)\,(TAIL\,y)))$$
$$\quad \textbf{and}$$
$$\quad\quad reverse\,x = reverse'\,x\ \textbf{nil}$$
$$\quad \textbf{and}$$
$$\quad\quad reverse'\,x\,z = IF\,(NULL\,x)\,z\,(reverse'\,(TAIL\,x)\,(PREFIX\,(HEAD\,x)\,z)).$$

The auxiliary functions *eqlist* and *reverse* can be readily expressed using $zip2'$ as

$$eqlist\,x\,y = zip2'\,x\ \textbf{true}\ AND\ EQUAL\,y$$
$$reverse\,x = zip2'\,x\ \textbf{nil}\ PREFIX\ \mathbf{K}\ DUMMY.$$

Note that the definition of *reverse* is derived from the one using $list2$ and the equation combining $list2$ and $zip2$. We have thus finished the generalization step. The final program can be obtained by substitution.

$$palindrome\,x = \xi\ \textbf{true}\ AND\ EQUAL\,(\xi\ \textbf{nil}\ PREFIX\ \mathbf{K}\ DUMMY)$$
$$\quad \textbf{where}\ \xi = zip2'\,x.$$

Lambda-hoisting of the function $zip2'$ has been completed in the definition of the type *'a list*.

4.2. Functions on Trees

We shall consider another data structure *'a tree*:

```
abstype 'a tree = TIP of 'a | FORK of ('a tree)('a tree)
  with
    val ISTIP(TIP _) = true | ISTIP(FORK _) = false
    and TIPVAL(TIP a) = a
    and LEFT(FORK lr) = l
    and RIGHT(FORK lr) = r
    and rec
      btree' x = Φ(IF(ISTIP x))(TIPVAL x)(btree'(LEFT x))
                 (btree'(RIGHT x))
        where Φβυηζgf = β(fυ)(g(ηgf)(ζgf))
end.
```

Here, the function *btree'* is based on a familiar form of tree traversal functions

```
btree gfx =
  IF(ISTIP x)(f(TIPVAL x))
  (g(btree gf(LEFT x))(btree gf(RIGHT x)))
```

that applies the function f to each tip of the tree x and combines the results of doing so on left and right subtrees by the function g over the tree.

We can solve the tree replacement problem in [1] using the traversal function *btree'*. The problem is to change a given binary tree into one identical in shape to the first, but with all the tip values replaced by the minimum of the tip values of the first. For example, the tree of Fig. 1 is changed into that of Fig. 2. (Figure 3 will be referred in Sect. 5.)

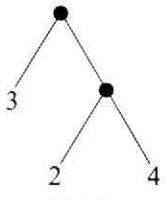

Fig. 1

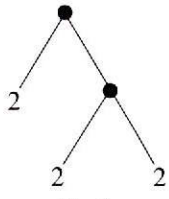

Fig. 2

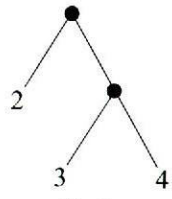

Fig. 3

A straightforward solution to this problem would be

```
transform x = replace x (tmin x)
  whererec
    replace xm =
      IF(ISTIP x)(TIP m)
      (FORK(replace(LEFT x)m)(replace(RIGHT x)m))
  and
    tmin x = IF(ISTIP x)(TIPVAL x)
             (MIN(tmin(LEFT x))(tmin(RIGHT x)))
```

where *MIN* yields the minimum of the two integers.

Table 1

Operation	Program		
	Straightforward	Ours	Bird's
IF	10	5	5
IF-TRUE	6	6	3
IF-FALSE	4	4	2
ISTIP	10	5	5
TIPVAL	3	3	3
LEFT	4	2	2
RIGHT	4	2	2
MIN	2	2	2
FORK	2	2	2
TIP	3	1	3
PAIR	-	-	5
FST	-	-	5
SND	-	-	5

We can define the auxiliary functions *replace* and *tmin* in terms of *btree'* as

$$replace\ x\ m = btree'\ x\ FORK\ (\lambda u.\ TIP\ m)$$
$$tmin\ x = btree'\ x\ MIN\ \mathbf{I}.$$

By substitution, we get

$$transform\ x = \xi\ FORK\ (\lambda u.\ TIP\ (\xi\ MIN\ \mathbf{I}))$$
$$\textbf{where}\ \xi = btree'\ x.$$

It is easy to see that this program traverses the tree x only once as in the case of list traversal. It is because our transformation technique does not depend on the data structure itself, but on the traversal function. There are many programs on tree structures that can be transformed similarly.

The original *transform* is of the *S*-composition form dealt with by Bird. We sketch here Bird's transformation method for comparison. To apply the method to this problem, we define a new function that produces a pair from a tree and an integer

$$repmin\ x\ m = PAIR\ (replace\ x\ m)\ (tmin\ x).$$

Once this function has been defined properly, the function can be rewritten as

$$transform\ x = FST\ p$$
$$\textbf{whererec}\ p = repmin\ x\ (SND\ p)$$

where $FST(PAIR\ a\ b) = a$ and $SND(PAIR\ a\ b) = b$. The reader should refer to [1] for the reasoning by which this has been derived from the definition of

repmin. The recursive definition of *repmin* can be obtained using the standard unfold-fold method [3].

$$
\begin{array}{l}
repmin\ x\ m = \\
\quad IF\ (ISTIP\ x)\ (PAIR\ (TIP\ m)\ (TIPVAL\ x)) \\
\quad (PAIR\ (FORK\ x1\ x2)\ (MIN\ m1\ m2)) \\
\qquad \textbf{whererec} \\
\qquad\quad x1 = FST\ r1\ \textbf{and}\ m1 = SND\ r1\ \textbf{where}\ r1 = repmin\ (LEFT\ x)\ m \\
\qquad \textbf{and} \\
\qquad\quad x2 = FST\ r2\ \textbf{and}\ m2 = SND\ r2\ \textbf{where}\ r2 = repmin\ (RIGHT\ x)\ m.
\end{array}
$$

Using this function, we can carry out the tip replacement in a single traversal of the tree.

Table 1 contains the number of primitive operations performed to print out the resulting tree of Fig. 2 when applying the function *transform* to the tree of Fig. 1. The effect of transformations is clearly seen from it.

5. Transformation Rules Revisited

We have excluded so far the case where the auxiliary traversal functions are mutually recursive. It is, however, too restrictive in practice as illustrated in the next example.

Consider a problem in [8]: Find the set of the deepest tips in a tree, where the set is represented by a list. A naive solution can be

$$
\begin{array}{l}
deepest\ x = \\
\quad IF\ (ISTIP\ x)\ (PREFIX\ (TIPVAL\ x)\ \textbf{nil}) \\
\qquad (IF\ (GREATER\ (depth\ l)\ (depth\ r))\ (deepest\ l) \\
\qquad\quad (IF\ (LESS\ (depth\ l)\ (depth\ r))\ (deepest\ r) \\
\qquad\qquad (APPEND\ (deepest\ l)\ (deepest\ r)))) \\
\qquad \textbf{whererec} \\
\qquad\quad l = LEFT\ x\ \textbf{and}\ r = RIGHT\ x \\
\qquad \textbf{and} \\
\qquad\quad depth\ x = \\
\qquad\qquad IF\ (ISTIP\ x)\ 0\ (PLUS\ 1\ (MAX\ (depth\ (LEFT\ x)) \\
\qquad\qquad (depth\ (RIGHT\ x)))).
\end{array}
$$

The function *APPEND* concatenates two lists and the function *MAX* gives the maximum of the two arguments. Here we can see that the function *deepest* does not only depend on itself, but also on the function *depth*. We have not yet established the rules for such cases.

There can be several possibilities of extending the idea of our transformation technique to deal with such cases. One way to do this would be to define general computational rules for the tuple $[g_1, \ldots, g_k]$ of the auxiliary functions. This may raise a new problem of defining the meaning of functional application $([g_1, \ldots, g_k]\ x)$ consistently. So, we shall deal with mutual recursion of the auxiliary functions differently. Although this approach is applicable to any data structure in general, we focus here on the tree structure.

Consider the general form of tree traversal programs.

$$
\begin{array}{l}
f\,x = F \\
\quad \textbf{whererec} \\
\quad\quad g_1\,x = IF\ (ISTIP\ x)\ (\psi_1\ (TIPVAL\ x))\ s_1 \\
\quad \textbf{and} \\
\quad\quad \ldots \\
\quad \textbf{and} \\
\quad\quad g_k\,x = IF\ (ISTIP\ x)\ (\psi_k\ (TIPVAL\ x))\ s_k .
\end{array}
$$

Each function s_i may contain $g_j (j \neq i)$ as well as g_i. We can assume here, however, that only the terms $(LEFT\ x)$ and $(RIGHT\ x)$ are the arguments of g_i and g_j in s_i, and no other terms do not contain the variable x. This implies that there is no term dependent on x directly nor indirectly except $(LEFT\ x)$ and $(RIGHT\ x)$. Since the auxiliary function g_i traverses the data structure x, such assumption is very natural in practice. Hence, we can define a higher order function common to all the g_i's using the following rules.

[Generalization with Lambda-hoisting]

Rewrite each s_i of the auxiliary function

$$g_i\,x = IF\ (ISTIP\ x)\ (\psi_i\ (TIPVAL\ x))\ s_i$$

as

$$
\begin{array}{l}
s_i = \sigma_i\ (g_i\ l)\ (g_i\ r) \\
\qquad (g_1\ l)\ (g_1\ r) \\
\qquad\quad \ldots \\
\qquad (g_{i-1}\ l)\ (g_{i-1}\ r) \\
\qquad DUMMY\ DUMMY \\
\qquad (g_{i+1}\ l)\ (g_{i+1}\ r) \\
\qquad\quad \ldots \\
\qquad (g_k\ l)\ (g_k\ r) \\
\quad \textbf{where} \\
\qquad l = LEFT\ x\ \textbf{and}\ r = RIGHT\ x
\end{array}
$$

where σ_i is a combinator, that is, a closed lambda term composed of only lambda variables and constants. The first pair of the arguments $(g_i\ l)$ and $(g_i\ r)$ corresponds to the self-recursive terms within s_i. The other arguments represent mutual recursive terms. Two $DUMMY$ arguments appear at i-th position for σ_i. This process can be done using a variant of the lambda-hoisting technique. The arity of the combinator is common to all the σ_i, and equal to $2k+2$.

Then the common recursive form can be written as

$$
\begin{array}{l}
h\,x\,\psi\,\sigma = \\
\quad IF\ (ISTIP\ x)\ (\psi\ (TIPVAL\ x)) \\
\qquad (\sigma\ (\eta\,\psi\,\sigma)\ (\zeta\,\psi\,\sigma) \\
\qquad\quad (\eta\,\psi_1\,\sigma_1)\ (\zeta\,\psi_1\,\sigma_1) \ldots (\eta\,\psi_k\,\sigma_k)\ (\zeta\,\psi_k\,\sigma_k)) \\
\qquad \textbf{where} \\
\qquad\quad \eta = h\ (LEFT\ x)\ \textbf{and}\ \zeta = h\ (RIGHT\ x).
\end{array}
$$

[Substitution]
Replace all the terms $(g_i x)$ in F by

$\xi\psi_i\sigma_i$ **where** $\xi = hx$

getting F'.

[Lambda-hoisting]
By applying the lambda-hoisting procedure to the function h above, we obtain the final form

$fx = F'$
where
$\xi = hx$
whererec
$hx = \Phi(IF(ISTIP\,x))(TIPVAL\,x)(h(LEFT\,x))(h(RIGHT\,x))$
where
$\Phi\beta v\eta\zeta\psi\sigma =$
$\beta(\psi v)$
$(\sigma(\eta\psi\sigma)(\zeta\psi\sigma)$
$(\eta\psi_1\sigma_1)(\zeta\psi_1\sigma_1)\ldots(\eta\psi_k\sigma_k)(\zeta\psi_k\sigma_k))$.

Although the function h can be defined in the abstract type *'a tree*, it would be less general than the higher order functions defined so far. Firstly, the function h is dependent on the number k of the auxiliary functions. And it is observed that only some part of the terms $(\eta\psi_i\sigma_i)$ and $(\zeta\psi_i\sigma_i)$ are actually needed as will be shown in the examples. In fact, redundant terms should not be included to keep the number of arguments as small as possible. Omission of unnecessary terms makes gains in execution time and space. Therefore, we will give a particular definition of h adapted for each problem.

We now try to transform the program *deepest* presented at the beginning of this section. Observe that the auxiliary functions *deepest* and *depth* can be rewritten as

deepest $x =$
$IF(ISTIP\,x)(\psi_1(TIPVAL\,x))$
$(\sigma_1(deepest\,l)(deepest\,r)(depth\,l)(depth\,r))$
where
$\psi_1 v = PREFIX\ v$ **nil**
and
$\sigma_1\eta\zeta\alpha_1\alpha_2 =$
$IF(GREATER\,\alpha_1\,\alpha_2)\,\eta(IF(LESS\,\alpha_1\,\alpha_2)\,\zeta(APPEND\,\eta\zeta))$
depth $x =$
$IF(ISTIP\,x)(\psi_2(TIPVAL\,x))$
$(\sigma_2(depth\,l)(depth\,r)\,DUMMY\,DUMMY)$
where
$\psi_2 v = 0$
and
$\sigma_2\eta\zeta\alpha_1\alpha_2 = PLUS\,1\,(MAX\,\eta\zeta)$

where $l=(LEFT\ x)$ and $r=(RIGHT\ x)$. We have assigned *deepest* to g_1 and *depth* to g_2. According to the general form of s_i, s_1 and s_2 in this case should have been

$$s_1=\sigma_1\,(deepest\ l)\,(deepest\ r)\,DUMMY\,DUMMY\,(depth\ l)\,(depth\ r)$$
$$s_2=\sigma_2\,(depth\ l)\,(depth\ r)\,(deepest\ l)\,(deepest\ r)\,DUMMY\,DUMMY.$$

But as mentioned earlier, s_2 does not contain any term on *deepest* and the arguments $(deepest\ l)$ and $(deepest\ r)$ of σ_2 are redundant. Hence we have omitted the third and fourth arguments of both σ_1 and σ_2. From the above definitions, we get the final program

```
deepest x = ξ ψ1 σ1
    where
        ξ = h x
            whererec
                h x = Φ (IF (ISTIP x)) (TIPVAL x) (h (LEFT x)) (h (RIGHT x))
                    where
                        Φ β υ η ζ ψ σ =
                            β (ψ υ) (σ (η ψ σ)) (ζ ψ σ) (η ψ2 σ2) (ζ ψ2 σ2))
                                where
                                    ψ2 υ = ... and σ2 η ζ α1 α2 = ...
    and
        ψ1 υ = ... and σ1 η ζ α1 α2 = ....
```

Our next example is again a tree replacement problem: Transform a binary tree into one of the same shape of which tip values are those of the original tree arranged in increasing order. For example, the tree of Fig. 1 is to be transformed into that of Fig. 3.

A solution with refinement by transformational programming is given in [1].

```
transform x = replace x (sort (tips x))
    whererec
        replace x z =
            IF (ISTIP x) (TIP (HEAD z))
               (FORK (replace (LEFT x) (take (size (LEFT x)) z))
                     (replace (RIGHT x) (drop (size (LEFT x)) z)))
    and
        size x = IF (ISTIP x) 1 (PLUS (size (LEFT x)) (size (RIGHT x)))
    and
        tips x = ntips x nil
    and
        ntips x z =
            IF (ISTIP x) (PREFIX (TIPVAL x) z)
               (ntips (LEFT x) (ntips (RIGHT x) z)).
```

The functions *take* and *drop* over list structures are defined as

```
take n z = IF (EQUAL n 0) nil (PREFIX (HEAD z)
               (take (MINUS n 1) (TAIL z)))
drop n z = IF (EQUAL n 0) z (drop (MINUS n 1) (TAIL z)).
```

Although there is much room for improvement on these functions, we confine ourselves to transformation for eliminating multiple traversals of the binary tree. We assume that an efficient sorting function *sort* exists.

Here, we illustrate the transformation process in another way; define first the function h, and then rewrite the auxiliary functions in terms of h. This is the way of programming with higher order generic functions. In this case, we need to include only the term $(size\,(LEFT\,x))$ as an argument to the combinator σ in the general form. Taking this into account, we have a common recursive form for the program *transform*

$$h\,x = \Phi\,(IF\,(ISTIP\,x))\,(TIPVAL\,x)\,(h\,(LEFT\,x))\,(h\,(RIGHT\,x))$$
where
$$\Phi\,\beta\,v\,\eta\,\zeta\,\psi\,\sigma =$$
$$\beta\,(\psi\,v)\,(\sigma\,(\eta\,\psi\,\sigma)\,(\zeta\,\psi\,\sigma)\,(\eta\,\psi_2\,\sigma_2))$$

where ψ_2 and σ_2 come from *size*.

Using the function h, we can rewrite the auxiliary functions:

$$replace\,x = h\,x\,\psi_1\,\sigma_1$$
where
$$\psi_1\,v = \lambda u.\,TIP\,(HEAD\,u)$$
and
$$\sigma_1\,\eta\,\zeta\,\alpha = \lambda u.\,FORK\,(\eta\,(take\,\alpha\,u))\,(\zeta\,(drop\,\alpha\,u))$$
$$size\,x = h\,x\,\psi_2\,\sigma_2$$
where
$$\psi_2\,v = 1$$
and
$$\sigma_2\,\eta\,\zeta\,\alpha = PLUS\,\eta\,\zeta$$
$$ntips\,x = h\,x\,\psi_3\,\sigma_3$$
where
$$\psi_3\,v = \lambda u.\,PREFIX\,v\,u$$
and
$$\sigma_3\,\eta\,\zeta\,\alpha = \lambda u.\,\eta\,(\zeta\,u).$$

By substitution we obtain the final program

$$transform\,x = \xi\,\psi_1\,\sigma_1\,(sort\,(\xi\,\psi_3\,\sigma_3\ \mathbf{nil}))$$
where
$$\xi = h\,x$$
whererec
$$h\,x = \Phi\,(IF\,(ISTIP\,x))\,(TIPVAL\,x)\,(h\,(LEFT\,x))\,(h\,(RIGHT\,x))$$
where
$$\Phi\,\beta\,v\,\eta\,\zeta\,\psi\,\sigma =$$
$$\beta\,(\psi\,v)\,(\sigma\,(\eta\,\psi\,\sigma)\,(\zeta\,\psi\,\sigma)\,(\eta\,\psi_2\,\sigma_2))$$
where
$$\psi_2\,v = \ldots \textbf{ and } \sigma_2\,\eta\,\zeta\,\alpha = \ldots$$
and
$$\psi_1\,v = \ldots \textbf{ and } \sigma_1\,\eta\,\zeta\,\alpha = \ldots$$
and
$$\psi_3\,v = \ldots \textbf{ and } \sigma_3\,\eta\,\zeta\,\alpha = \ldots.$$

The transformation rules above can be applied in a systematic way; no heuristic process employed.

6. Conclusion

Although our transformation technique has been applied to small programs in this paper, there is no problems encountered on applying it to practical programs if we establish the transformation rules for the data structure that the program deals with. Examples of such rules have been shown in Sects. 4 and 5. The transformation rules for various kind of data structures lead to the possibility of an automatic transformation system for optimizing functional programs. Transformed programs have to be evaluated in a lazy way; the argument expression is never evaluated until required, and when it is evaluated the argument is replaced by its result. Hence a partially parametrized function shared by several positions in an expression effectively distributes information obtained by the first visit to every data item. In fact it is memo-ized as a function closure which is bound to the shared variable corresponding to the argument.

It should be noted that our technique imposes no restriction on the forms of functional composition; Bird [1] deals with the form $fx(gx)$. As shown by the examples, our rules do not depend on the form of functional composition, but do only on the traversal function. As for the language issue, it is assumed that our functional language allows the higher order function and its partial parametrization in addition to lazy evaluation and local definition mechanisms assumed by Bird. In fact these features, including partial parametrization of higher order functions, are particularly powerful and useful tools in functional programming. We hope that such novel features in functional programming will be investigated from various approaches.

Acknowledgements. I sincerely thank Kohei Noshita for continuing encouragement in the early stages of this research, and also the referees for their detailed comments and suggestions.

References

1. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Inf. **21**, 239–250 (1984)
2. Burge, W.H.: Recursive Programming Techniques. Reading, MA: Addison-Wesley 1975
3. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**, 44–67 (1977)
4. Friedman, D.P., Wise, D.S.: CONS should not evaluate its arguments. Proc. 3rd International Colloquium on Automata, Languages and Programming, pp. 257–284. Edinburgh (1976)
5. Henderson, P., Morris, J.M.: A lazy evaluator. Proc. 3rd Symp. on Principle of Programming Languages, pp. 95–103. Atlanta, GA (1976)
6. Hughes, R.J.M.: The design and implementation of programming languages. D.Phil. thesis. Oxford University 1984
7. Hughes, R.J.M.: Super-combinators: a new implementation method for applicative languages. Proc. 1982 ACM Symp. Lisp and Functional Programming, pp. 1–10. Pittsburgh, PA (1982)

8. Hughes, R.J.M.: Lazy memo-functions. Functional Programming Languages and Computer Architecture. Lect. Notes Comp. Sci. 201, pp. 129–146. Berlin, Heidelberg, New York: 1985
9. Johnsson, T.: Lambda-lifting: Transforming Programs to Recursive Equations. *ibid.*, 190–203 (1985)
10. Milner, R.: A proposal for Standard ML. Proc. 1984 ACM Symp. LISP and Functional Programming, pp. 184–197. Austine, TX (1984)
11. Turner, D.A.: Aspects of the implementation of programming languages. D.Phil. thesis. Oxford University 1981
12. Vuillemin, J.: Correct and optimal implementations of recursion in a simple programming language. J. Comp. Syst. Sci. **9**, 332–354 (1974)
13. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda-Calculus. D.Phil. thesis. Oxford University 1971

Received November 22, 1985/August 28, 1986