Vol. 42 No. SIG 7(PRO 11) IPSJ Transactions on Programming July 2001

Regular Paper

Recursion Removal and Introduction Using Assignments

Kazuhiko Kakehi,[†,☆] Robert Glück[††,☆☆]
and Yoshihiko Futamura[†††,☆☆]

Recursive programs are often easy to write and reason about, while iterative ones are usually more efficient to execute. Transformation between recursive and iterative variants of a function is therefore important in order to enjoy the benefits of both programming styles. Recursion removal has been energetically researched for many years. In case of functions which consume a list and produce a list, however, recursion removal is not so straightforward without introduction of auxiliary stacks. We first define abstraction from constructors and some kinds of constructing functions, and propose a recursion removal method from constructing functions. This method produces tail-recursive programs from linear recursive functions with accumulation of abstracted expressions. With specialization using partial evaluation techniques, the interpretive overhead of constructor abstraction can be eliminated and fast execution is realized. This technique works not only for linear recursion but also for tree recursive functions or certain forms of nested functions. The idea of interpretation of abstracted construction enables not only recursion removal but also elimination of accumulating parameters. Transformed functions without accumulating parameters are executed in recursive manner, and recursion is introduced to such functions. This paper intends to present the way of these kinds of recursion removal and introduction as well as the representation of abstracted constructions which enables these program transformations.

1. Introduction

Recursive programs are often easy to write and reason about[16),48)], while iterative ones are usually more efficient to execute. Transformation between recursive and iterative variants of a function is therefore important in order to enjoy the benefits of both programming styles. Recursion removal has been energetically researched for many years. In case of functions which translate lists, however, recursion removal is not straightforward without introduction of auxiliary stacks, or help of associativity of `append`, and the knowledge of associativity cannot always be discovered automatically. This is because constructors of lists lack associativity.

Structures consist of constructors as containers and their contents including pointers. Since their evaluation order does not matter with the assumption that there are no side-effects, we focus on destructive operations, like `rplacd` in Lisp or `set-cdr!` in Scheme. With the help of such operations, we first introduce abstraction from constructors and some kinds of constructing functions which we call functional constructors. Using this abstraction we then propose a recursion removal method from functions which produce a structure. This translation basically intends for linear recursive functions with construction. Tree recursive functions using constructors and some kinds of nested functions can also be dealt with by our method.

Abstracted expressions is handled and evaluated by interpretive manners. Using partial evaluation techniques, such interpretive overhead is eliminated by specialization and fast execution of the iterative variants is realized.

The idea of interpretation of abstracted expressions enables not only recursion removal but also other areas. This abstraction eliminates an accumulating parameter and the new functions are executed in a recursive manner. This transformation introduces recursion to certain forms of iterative programs.

The rest of this paper is organized as follows. Section 2 explains basic ideas of recursion removal, and Section 3 gives an overview of our idea to remove recursion. Section 4 investigates abstraction from constructing expressions including certain kinds of functions. This ab-

† Graduate School of Science and Engineering, Waseda University
†† PRESTO, Japan Science and Technology Corporation
††† School of Science and Engineering, Waseda University
☆ Presently with Research Fellow of the Japan Society for the Promotion of Science
☆☆ Presently with Institute for Software Production Technology, Waseda University

```
define linear(x)
 case p(x) of
      True  → b(x)
      False → a(c(x), linear(d(x)))
```

a: auxiliary function b: base function
c: control function d: descent function
p: termination condition

Fig. 1 Definition skeleton of right linear recursion.

straction enables us to have associativity in constructing expressions with cheap expense. Section 5 is the core part on recursion removal. Using the idea of abstraction, transformation is done in two steps of accumulation and specialization. Section 6 demonstrates transformation steps in detail and Section 7 measures how much effect can be obtained by our idea. Section 8 explains other aspects of our abstraction method, including recursion introduction. Section 9 compares with other related works, and Section 10 concludes with mentioning future works.

2. Basic Ideas of Recursion Removal

In functional programming, functions are usually expressed in recursive forms. Execution of recursive functions requires new stack frames, except for tail recursion which is equivalent to iteration, and manipulation of stacks makes programs slower. Recursion removal is a program transformation to obtain functionally equivalent programs with reduced number of stacks. It is generally analyzed by human and used in order to gain speedup.

Due to its importance, recursion removal has been researched for a long time[12]. For example, the oldest literature we found which mentions the relation between recursion and iteration appears in 1963[9], and some kind of general transformation appeared in 1966[17]. Despite the importance and history it has, however, implementing recursion removal, even for linear recursive functions, into real compilers is rare. As one of evidence, compiler books do not include topics of recursion removal except for tail recursion[2),5),49]. This is because the transformation is not yet ripe for full automation.

Now we observe basic approaches of recursion removal. Linear recursive functions are defined as functions having at most one recursive call to itself in each branch. Right linear recursion, one style of linear recursion, is illustrated in **Fig. 1**.

If we remove recursion, one simple method exists using an auxiliary accumulator and counter[35]. That is, first we find the minimum $\mathtt{n}$ which fulfills $\mathtt{p}(\mathtt{d}^{\mathtt{n}}(\mathtt{x}))$, and store $\mathtt{n}-1$ in the counter $\mathtt{m}$. We then calculate $\mathtt{b}(\mathtt{d}^{\mathtt{n}}(\mathtt{x}))$ and store it into the accumulator $\mathtt{acc}$, and repeatedly compute $\mathtt{a}(\mathtt{c}(\mathtt{d}^{\mathtt{m}}(\mathtt{x})), \mathtt{acc})$ for decrementing $\mathtt{m}$ until $\mathtt{m} = 0$. This method only requires two additional parameters, but needs $O(\mathtt{n}^2)$ computation for $\mathtt{d}$. As far as recursion removal targets for faster execution, this solution is not sufficient. We need restriction that the computational complexity should not be worsened by this program transformation.

Recursion removal methods toward linear recursion, without worsening complexity, are categorized into two.

(1) The first method is similar to what we have seen: tracing how the input is decremented without calculating the result at this moment, and start calculation from a value which suffices a termination condition $\mathtt{p}$ to the given input. The trouble we face is that we need to trace back the descent of input. To avoid this inefficiency, existence of an inverse of descent functions $\mathtt{d}^{-1}$, or auxiliary stacks to store the history of input is sufficient. Since the computation starts from a terminating branch and goes back to outer calculation toward original input, we call this technique "*inside-out* manner".

(2) The other method is to obtain iterative variants of recursive programs by calculating output gradually, following the original descent of recursion parameters. Computation starts from the given input and finishes when it reaches some terminating condition. We here call this "*outside-in* manner". In outside-in recursion removal, we do not need the inverse or auxiliary stacks, because input is traced only once. Auxiliary stack is just a substitute for stack frames which are needed for executing recursive procedures, so this is one big benefit of outside-in recursion removal.

In this paper, we pursue a recursion removal method based on the outside-in idea. This type of techniques, however, requires other analyses to fulfill the transformation. The following two subsections demonstrate typical techniques of outside-in transformations.

2.1 Associativity

One technique is to investigate associativity of auxiliary functions $\mathtt{a}$. Factorial function, for example, is defined for nonnegative integer $\mathtt{x}$:

$$\begin{array}{l}\mathtt{define\ fact(x)\ case\ (x=0)\ of}\\ \quad \mathtt{True} \quad \rightarrow \mathtt{1}\\ \quad \mathtt{False} \quad \rightarrow \mathtt{x} \times \mathtt{fact(x-1)}\end{array}$$

By defining a new function $\mathtt{fact}'$ as

$$\mathtt{fact}'(\mathtt{y},\mathtt{x}) = \mathtt{y} \times \mathtt{fact}(\mathtt{x}),$$

we obtain the body of $\mathtt{fact}'$ by following transformations:

$$\begin{array}{l}\mathtt{fact}'(\mathtt{y},\mathtt{x}) = \mathtt{y} \times \mathtt{fact}(\mathtt{x})\\ \textit{(unfolding } \mathtt{fact}\textit{)}\\ \Rightarrow \mathtt{y} \times (\mathtt{case\ (x=0)\ of}\\ \qquad \mathtt{True} \quad \rightarrow \mathtt{1}\\ \qquad \mathtt{False} \quad \rightarrow \mathtt{x} \times \mathtt{fact(x-1)})\\ \textit{(distribution)}\\ \Rightarrow \mathtt{case\ (x=0)\ of}\\ \qquad \mathtt{True} \quad \rightarrow \mathtt{y} \times \mathtt{1}\\ \qquad \mathtt{False} \quad \rightarrow \mathtt{y} \times (\mathtt{x} \times \mathtt{fact(x-1)})\\ \\ \textit{(partial evaluation of } \times 1\textit{; associativity)}\\ \Rightarrow \mathtt{case\ (x=0)\ of}\\ \qquad \mathtt{True} \quad \rightarrow \mathtt{y}\\ \qquad \mathtt{False} \quad \rightarrow (\mathtt{y} \times \mathtt{x}) \times \mathtt{fact(x-1)}\\ \textit{(folding to } \mathtt{fact}'\textit{)}\\ \Rightarrow \mathtt{case\ (x=0)\ of}\\ \qquad \mathtt{True} \quad \rightarrow \mathtt{y}\\ \qquad \mathtt{False} \quad \rightarrow \mathtt{fact}'(\mathtt{y} \times \mathtt{x}, \mathtt{x}-\mathtt{1})\end{array}$$

Giving multiplication unit 1 to $\mathtt{y}$, we obtain a tail recursive variant of $\mathtt{fact}$. This analysis appears very often and commonly used for recursion removal. As we see, associativity of the auxiliary function, multiplication in this case, enables transformation. Transformations based on associativity, in turn, suffer from the fact that its investigation is not an easy task. Moreover, there are many functions which lack associativity. Constructors like $\mathtt{Cons}$ are good examples. This technique fails for $\mathtt{append}$ in Fig. 4, because: $\mathtt{Cons}[\mathtt{Cons}[\mathtt{x1},\mathtt{x2}],\mathtt{x3}] \neq \mathtt{Cons}[\mathtt{x1},\mathtt{Cons}[\mathtt{x2},\mathtt{x3}]]$.

2.2 Lambda Abstraction

Another way to realize removal of recursion in the same manner is lambda abstraction. Church-Rosser property enables transformation of any linear functions. Similar to the case of recursion removal using associativity, we define new function $\mathtt{linear}'(\mathtt{z},\mathtt{x}) = \mathtt{z}\,(\,\mathtt{linear}(\mathtt{x})\,)$, where $\mathtt{z}$ is a λ-term and application of an expression *exp* to $\mathtt{z}$ is denoted by $\mathtt{z}\,(\mathit{exp})$. The body of $\mathtt{linear}'$ is translated as:

$$\begin{array}{l}\mathtt{linear}'(\mathtt{z},\mathtt{x}) = \mathtt{z}\,(\,\mathtt{linear(x)}\,)\\ \textit{(unfolding } \mathtt{linear}\textit{)}\\ \Rightarrow \mathtt{z}\ \ (\ \mathtt{case\ p(x)of}\\ \qquad \mathtt{True} \ \ \rightarrow \mathtt{b(x)}\\ \qquad \mathtt{False} \rightarrow \mathtt{a(c(x),linear(d(x)))}\ )\\ \textit{(}\lambda\textit{-abstraction)}\\ \Rightarrow \mathtt{z}\ \ (\ \mathtt{case\ p(x)of}\\ \qquad \mathtt{True} \ \ \rightarrow \mathtt{b(x)}\\ \qquad \mathtt{False} \rightarrow (\lambda i.\mathtt{a}(\mathtt{c(x)}, i))\\ \qquad\qquad\qquad\qquad (\ \mathtt{linear(d(x))}\ )\ )\\ \textit{(distribution)}\\ \Rightarrow \ \mathtt{case\ p(x)of}\\ \quad \mathtt{True} \ \ \rightarrow \mathtt{z}\,(\ \mathtt{b(x)}\ )\\ \quad \mathtt{False} \rightarrow \mathtt{z}\ (\ (\lambda i.\mathtt{a}(\mathtt{c(x)}, i))\\ \qquad\qquad\qquad\qquad (\ \mathtt{linear(d(x))}\ )\ )\\ \textit{(Church-Rosser)}\\ \Rightarrow \ \mathtt{case\ p(x)\ of}\\ \quad \mathtt{True} \ \ \rightarrow \mathtt{z}(\ \mathtt{b(x)}\ )\\ \quad \mathtt{False} \rightarrow (\mathtt{z}\ (\lambda i.\mathtt{a}(\mathtt{c(x)}, i)))\\ \qquad\qquad\qquad\qquad (\ \mathtt{linear(d(x))}\ )\\ \textit{(folding to } \mathtt{linear}'\textit{)}\\ \Rightarrow \ \mathtt{case\ p(x)\ of}\\ \quad \mathtt{True} \ \ \rightarrow \mathtt{z}(\ \mathtt{b(x)}\ )\\ \quad \mathtt{False}\\ \qquad \rightarrow \mathtt{linear}'(\mathtt{z}\ (\lambda i.\mathtt{a}(\mathtt{c(x)}, i)), \mathtt{d(x)})\end{array}$$

Since identity $\lambda i.i$ functions as an unit term, we have a tail recursive definition of $\mathtt{linear}$ by giving $\lambda i.i$ to $\mathtt{z}$ as an initial value. As far as Church-Rosser property holds, this transformation is possible to any linear functions. Despite its usefulness, however, the expression and calculation of lambda terms, namely closures are expensive operations, and this transformation may not achieve the desired optimization.

3. Overview of Our Approach

In the previous section we have seen techniques to realize recursion removal from linear recursion. Constructors are common for representing structures of statically unbound size, and it is important to remove recursion from constructing functions. The techniques shown in the previous section are not sufficient for such constructing functions.

We now describe how to remove recursion from constructing functions. For simplicity, we use a simple functional language with pattern matching. Syntax and semantics are defined in **Figs. 2** and **3**, respectively. The three running examples in this paper are given in **Fig. 4**. Before investigating methods for 'outside-in' recursion removal toward constructing functions, we turn our eyes on delayed initialization of con-

$$
\begin{array}{rcll}
p & ::= & d+ & \text{— program} \\
d & ::= & \texttt{define}\ f(vr+, vc*)\ b & \text{— definition} \\
b & ::= & \texttt{case}\ t\ \texttt{of}\ \{pat \rightarrow b\}+ & \text{— body} \\
 & | & e & \\
e & ::= & v \mid a \mid c[e*] \mid f(e+) & \text{— expression} \\
t & ::= & vr \mid a \mid ope(t+) & \text{— term} \\
pat & ::= & c[pat*] \mid v \mid a & \text{— pattern}
\end{array}
$$

$a \in$ constants like *Int*, ... $\quad$ $v \in$ *variable names* where
$c \in$ *constructor names* $\quad$ vr: recursion parameters
$f \in$ *function names* $\quad$ vc: context parameters
ope : built-in operations

Constructors take contents enclosed with square brackets with the constructor name; if a constructor takes no arguments, as is the case in `Nil` or boolean values, we omit square brackets. Functions, on the other hand, uses ordinary round brackets, like `f(x, y)`.

Fig. 2 Source language p.

Standard semantics:

$$\frac{}{\sigma, prog \vdash_{sem} a \Rightarrow [\![a]\!]} \qquad \frac{}{\sigma, prog \vdash_{sem} v \Rightarrow \sigma[v]}$$

$$\frac{\sigma, prog \vdash_{sem} e_i \Rightarrow e'_i \quad e'_i \text{ is not a } aterm \text{ for } \forall i}{\sigma, prog \vdash_{sem} c[e_1 ... e_m] \Rightarrow [\![c]\!][e'_1 ... e'_m]} \qquad \frac{\sigma, prog \vdash_{sem} t_i \Rightarrow t'_i \text{ for } \forall i}{\sigma, prog \vdash_{sem} ope(t_1 ... t_m) \Rightarrow [\![ope]\!](t'_1 ... t'_m)}$$

$$\frac{\begin{array}{l}\sigma, prog \vdash_{sem} e_i \Rightarrow e'_i \text{ for } \forall i \\ prog[f] = \texttt{define}\ f(v_1 ... v_m)\ b \\ \sigma' = \sigma[v_1 \rightarrow e'_1 ... v_m \rightarrow e'_m] \\ \sigma', prog \vdash_{sem} b \Rightarrow out\end{array}}{\sigma, prog \vdash_{sem} f(e_1 ... e_m) \Rightarrow out} \qquad \frac{\begin{array}{l}\sigma, prog \vdash_{sem} t \Rightarrow t' \\ pat\text{-}match(\sigma, t', pat_1 ... pat_m) = [i, \sigma'] \\ \sigma', prog \vdash_{sem} b_i \Rightarrow out\end{array}}{\begin{array}{l}\sigma, prog \vdash_{sem} \texttt{case}\ t\ \texttt{of} \\ \quad \{pat_1 \rightarrow b_1\ ...\ pat_m \rightarrow b_m\} \Rightarrow out\end{array}}$$

- The operation *pat-match* does pattern matching between t and patterns $pat_1 ... pat_m$, returns branch number i and updated environment σ'.

Fig. 3 Semantics for the source language defined in Fig. 2.

```
define append(x, y) case x of
  Nil          → y
  Cons[x1, xs] → Cons[x1, append(xs, y)]

define lflat(x) case x of
  Nil          → Nil
  Cons[x1, xs] → append(x1, lflat(xs))

define flip(x) case x of
  Leaf[n]     → Leaf[n]
  Node[l, r]  → Node[flip(r), flip(l)]
```

Fig. 4 Three examples.

structing expressions, and give the overview of our approach.

3.1 Delayed Initialization

Taking `Cons` as an example, its evaluation☆ in call-by-value semantics is like:

(1) evaluate the expression in the `car` part,
(2) evaluate the expression in the `cdr` part,
(3) make a `Cons` cell using evaluated values.

Assume there are no side-effects, then the evaluation order does not matter. Constructors are just boxes, or containers, to hold values

☆ We use *eval* to evaluate the expression, and the semantic values are denoted by $[\![\ ]\!]$, like $[\![\texttt{Cons}]\!]$.

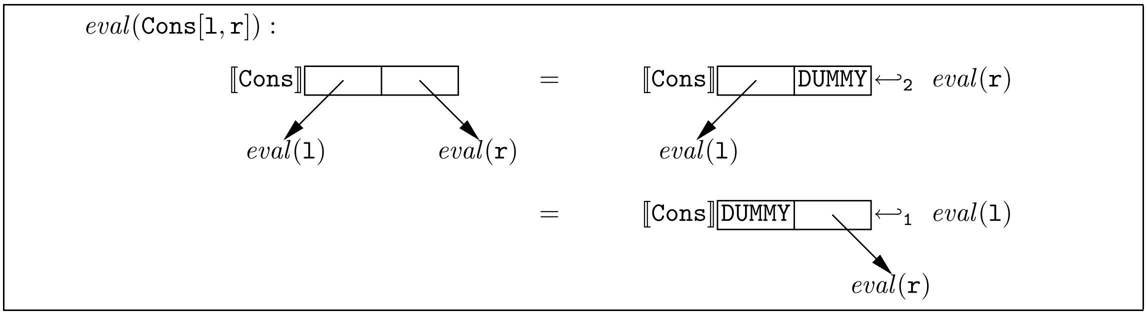


Fig. 5 Constructors as container boxes.

or pointers inside, and once constructors themselves are allocated, the values inside can be later assigned into constructors. For example, the evaluation of `cdr` part can be delayed, and later the delayed value be initialized by destructive assignments.

We use an infix operator $\hookleftarrow_i$, which does an assignment into a constructor cell allocated in the heap. The semantics of *left* $\hookleftarrow_i$ *right* is, for an allocated cell pointed by the pointer *left* and an evaluated value *right*,

(1) assign the right value *right* into the position i of left constructor pointed by *left*,

(2) return the pointer *left*.

Using such assignments, evaluation of an expression which appears at a position i in the construction can be delayed, while the construction itself takes its shape. We here use a constructor `DUMMY` for a place holder which fills the hole left unevaluated and is later initialized using these assignments (**Fig. 5**).

3.2 Transformation Strategy

Indeed we can delay initialization in constructors using assignment operations, they are basically operations with side-effects. In order to achieve ease of analysis and safety on semantics, the analysis proceeds in two steps.

(1) We first extend our language with abstraction from expressions which construct a structure. Section 4 explains the ideas and properties of the extension. Its syntax and evaluation semantics are given in Figs. 6 and 7, respectively.

Using this extended language, recursive programs are translated into iteration using accumulators. The transformation is explained in Section 5.1, and the rules are given by Figs. 8 and 9. This transformation reduces stack usages, but interpretation on the extended language is yet required.

(2) Translated programs in the extended language are specialized directly to use assignment operations. Section 5.2 explains its ideas.

4. Abstraction from Constructors and Functional Constructors

This section extends our source language in Fig. 2 to include abstraction from constructing expressions. This extension includes $\langle\!\langle\ \rangle\!\rangle$-expressions to denote abstracted expressions which construct a structure, and their evaluation results in $\langle\ \rangle$-terms.

Syntax and semantics of the language extended with $\langle\!\langle\ \rangle\!\rangle$-expressions are given in **Figs. 6** and **7**. We will only consider well-formed expressions $\langle\!\langle\ \rangle\!\rangle$-expressions as described in the following subsections.

4.1 Abstraction from Constructors

The analysis in Section 3.1 showed that assignment operations enable abstraction from constructors. In order to make assignments implicit, We use ($\langle\!\langle\ \rangle\!\rangle$) for denoting abstraction from constructors. Inside of $\langle\!\langle\ \rangle\!\rangle$ there appear two kinds of information: (1) an expression to create a structure, and (2) the position describing where an uninitialized hole exists. The hole is filled with `DUMMY`. This position number is given in Dewey notation[28].

As an example, we take an expression $\mathtt{Cons}[\mathtt{Cons}^1[\mathtt{x1}^{11}, \mathtt{f(x2)}^{12}], \mathtt{Cons}^2[\mathtt{f(x3)}^{21}, \mathtt{x4}^{22}]]$ where we added the position number as a superscript. If we abstract $\mathtt{f(x3)}^{21}$ from it, the abstracted expression becomes
$\langle\!\langle \mathtt{Cons}[\mathtt{Cons}[\mathtt{x1}, \mathtt{f(x2)}], \mathtt{Cons}[\mathtt{DUMMY}, \mathtt{x4}]], \mathtt{21} \rangle\!\rangle$.
We call this a $\langle\!\langle\ \rangle\!\rangle$-expression. This is a lambda abstraction adapted toward construction. Following lambda abstraction, an application to a $\langle\!\langle\ \rangle\!\rangle$-expression, namely delayed initializations, is expressed using neighboring sequence. Therefore
$\langle\!\langle \mathtt{Cons}[\mathtt{Cons}[\mathtt{x1}, \mathtt{f(x2)}], \mathtt{Cons}[\mathtt{DUMMY}, \mathtt{x4}]], \mathtt{21} \rangle\!\rangle \mathtt{f(x3)}$

$$
\begin{array}{lcl}
pe & ::= & d+ \cup de* \\
de & ::= & \texttt{define}\ f(vr+, vc*, va*)\ be \\
be & ::= & \texttt{case}\ t\ \texttt{of}\ \{pat \rightarrow be\}+ \\
 & | & exps
\end{array}
\qquad
\begin{array}{lcl}
exps & ::= & e \mid exp \mid exp\ exps \\
exp & ::= & va \mid \langle\!\langle ei, pos \rangle\!\rangle \\
 & | & c[exps*] \mid f(exps+) \\
ei & ::= & v \mid a \mid c[ei*] \mid f[ei+] \mid \texttt{DUMMY}
\end{array}
$$

exp :	expressions and abstracted expressions	*exps* :	sequence of expressions
ei :	expressions in abstracted expressions	*pos* :	position number
va :	parameters for abstracted values		

Fig. 6 Extended language *pe*.

Semantics extension:

$$\frac{\sigma, prog \vdash_{abst} ei \Rightarrow out \qquad \vdash_{chk} out \Rightarrow out'}{\sigma, prog \vdash_{sem} \langle\!\langle ei, pos \rangle\!\rangle \Rightarrow out'}$$

$$\frac{str = \texttt{DUMMY}}{\vdash_{chk} [str, 1, [\,]\,] \Rightarrow \langle\!\langle \texttt{DUMMY}, \texttt{0} \rangle\!\rangle} \qquad \frac{str \neq \texttt{DUMMY}}{\vdash_{chk} [str, flg, loc] \Rightarrow \langle str, loc \rangle}$$

Abstraction rules:

$$\frac{}{\sigma, prog \vdash_{abst} \texttt{DUMMY} \Rightarrow [\texttt{DUMMY}, 1, [\,]\,]}$$

$$\frac{}{\sigma, prog \vdash_{abst} a \Rightarrow [[\![a]\!], 0, [\,]\,]}$$

$$\frac{}{\sigma, prog \vdash_{abst} v \Rightarrow [\sigma[v], 0, [\,]\,]}$$

$$\frac{\begin{array}{l}\sigma, prog \vdash_{abst} ei_i \Rightarrow [ei'_i, flg_i, loc_i] \quad \text{for } \forall i \\ tmp = [\![c]\!][ei'_1 ... ei'_m] \\ loc' = loc_1 \mathbin{+\!\!+} ... \mathbin{+\!\!+} loc_m \\ loc'' = take1([[1, flg_1] ... [m, flg_m]], tmp)\end{array}}{\begin{array}{l}\sigma, prog \vdash_{abst} c[ei_1 ... ei_m] \\ \qquad \Rightarrow [tmp, 0, loc' \mathbin{+\!\!+} loc'']\end{array}}$$

$$\frac{\begin{array}{l}\sigma, prog \vdash_{abst} ei_i \Rightarrow [ei'_i, flg_i, loc_i] \quad \text{for } \forall i \\ prog[f] = \texttt{define}\ f(v_1 ... v_m)\ b \\ \sigma' = \sigma[v_1 \rightarrow ei'_1 ... v_m \rightarrow ei'_m] \\ \sigma', prog \vdash_{abst} b \Rightarrow [str, flg, loc] \\ loc' = loc_1 \mathbin{+\!\!+} ... \mathbin{+\!\!+} loc_m \mathbin{+\!\!+} loc\end{array}}{\sigma, prog \vdash_{abst} f(ei_1 ... ei_m) \Rightarrow [str, flg, loc']}$$

$$\frac{\begin{array}{l}\sigma, prog \vdash_{sem} t \Rightarrow t' \\ pat\text{-}match(\sigma, t', pat_1 ... pat_m) = [i, \sigma'] \\ \sigma', prog \vdash_{abst} b_i \Rightarrow out\end{array}}{\begin{array}{l}\sigma, prog, \vdash_{abst} \texttt{case}\ t\ \texttt{of} \\ \quad \{pat_1 \rightarrow b_1\ ...\ pat_m \rightarrow b_m\} \Rightarrow out\end{array}}$$

Application rules:

$$\frac{\begin{array}{l}\sigma, prog \vdash_{sem} exps \Rightarrow exps' \\ \sigma, prog \vdash_{sem} aexp \Rightarrow aexp' \\ \vdash_{appl} [aexp', exps'] \Rightarrow out\end{array}}{\sigma, prog \vdash_{sem} aexp\ exps \Rightarrow out}$$

$$\frac{}{\vdash_{appl} [\langle str, [\,] \rangle, term] \Rightarrow str}$$

$$\frac{}{\vdash_{appl} [\langle str, [\,] \rangle, aterm] \Rightarrow \langle str, [\,] \rangle}$$

$$\frac{}{\vdash_{appl} [\langle\!\langle \texttt{DUMMY}, \texttt{0} \rangle\!\rangle, exp'] \Rightarrow exp'} \qquad \frac{}{\vdash_{appl} [aexp', \langle\!\langle \texttt{DUMMY}, \texttt{0} \rangle\!\rangle] \Rightarrow aexp'}$$

$$\frac{ptr_i \hookleftarrow_{pos_i} exp \quad \text{for each } [ptr_i, pos_i] \text{ in } loc}{\vdash_{appl} [\langle str, loc \rangle, exp] \Rightarrow str} \qquad \frac{ptr_i \hookleftarrow_{pos_i} exp \quad \text{for each } [ptr_i, pos_i] \text{ in } locl}{\vdash_{appl} [\langle strl, locl \rangle, \langle strr, locr \rangle] \Rightarrow \langle strl, locr \rangle}$$

- *take1* takes two parameters *lst* and *ptr*. For each element $[i, flg_i]$ in *lst*, it returns a list of $[ptr, i]$ where $flg_i = 1$, e.g., $take1([[1,1],[2,0],[3,1]], tmp) = [[tmp, 1], [tmp, 3]]$.
- *aterm* ranges over $\langle\ \rangle$-terms including $\langle\!\langle \texttt{DUMMY}, \texttt{0} \rangle\!\rangle$.
- *term* ranges over evaluated terms except for $\langle\ \rangle$-terms.
- *aexp* ranges over any expressions which returns an *aterm*.

Fig. 7 Semantics for the extended language.

$= \texttt{Cons}[\texttt{Cons}[\texttt{x1}, \texttt{f}(\texttt{x2})], \texttt{Cons}[\texttt{f}(\texttt{x3}), \texttt{x4}]]$.

These $\langle\!\langle\ \rangle\!\rangle$-expressions are evaluated and constructor cells are allocated in the heap memory. In addition we need to know where abstracted values, currently represented by DUMMY, exist in the construction. We will use single angle brackets ($\langle\ \rangle$) to denote a concrete structure containing DUMMY which is already allocated in heap memory. Similar to $\langle\!\langle\ \rangle\!\rangle$-expressions, $\langle\ \rangle$ holds two informations: (1) pointer to the top of construction allocated in the heap memory, and (2) location information where DUMMY in the construction appears. This location information is a list of tuples, the pointer to the concrete parent constructor of the hole and the position in the constructor in Dewey notation.

In the above example, $\langle\!\langle \texttt{Cons}[\texttt{Cons}[\texttt{x1}, \texttt{f}(\texttt{x2})], \texttt{Cons}[\texttt{DUMMY}, \texttt{x4}]], 21\rangle\!\rangle$ is evaluated to $\langle str, loc\rangle$, where str is the pointer to the structure made by evaluating $\texttt{Cons}[\texttt{Cons}^1[\texttt{x1}^{11}, \texttt{f}(\texttt{x2})^{12}], \texttt{Cons}^2[\texttt{DUMMY}^{21}, \texttt{x4}^{22}]]$, and loc holds a tuple of information: the pointer to the cell allocated by the inner right $\texttt{Cons}^2$ and the position number 1. When applications take place, the location information loc is used for assignments.

A special case is $\langle\!\langle \texttt{DUMMY}, 0\rangle\!\rangle$. In Dewey notation 0 is not used, but we use 0 for pointing to the root position in a tree structure. This $\langle\!\langle\ \rangle\!\rangle$-expression means that the abstracted value appears on the location it is abstracted from, namely the position itself. Therefore this matches to identity $\lambda i.i$ in lambda terms. Since this cannot be 'allocated' in heap memory, this is kept as it is and an interpreter or compiler takes care of it.

Our notations show that we can implement closure-like structure for constructors with the help of delaying initialization using assignment operators.

4.2 Abstraction from Functional Constructors

We have seen that constructors can enjoy advantage of taking shape without initializing values inside. Functions, in general, cannot have that advantage since the evaluation of functions needs all parameters. But interesting exceptions exist. In case they are functions that build data structures, some of them can take the same advantage as constructors.

In our language settings in Fig. 2, we separate context parameters from recursion parameters. Recursion parameters are regarded as ones decomposed by `case` expressions, and their values have to be known at that point to proceed execution further. Context parameters, on the other hand, need not to be known when branching takes place. This property of context parameter holds the same characteristics as constructors: Constructors are just boxes with holes which hold values or pointers, and these values are reflected in the output but not necessarily known when the constructions are made, thanks to delayed initialization. Functions with context parameters can be regarded as structures depending on the status of their recursion parameters. We name such functions as 'functional constructors' with respect to the context parameters. The restriction imposed for this functional constructors is explained in the next subsection.

Now that abstraction of context parameters from functional constructors are safe, we can make $\langle\!\langle\ \rangle\!\rangle$-expressions from functional constructors. As a simple example, `append` has one recursion parameter `x` and one context parameter `y`, as shown in Fig. 4. Using double angle brackets, the second parameter `y` is abstracted out from `append` and it leaves $\langle\!\langle \texttt{append}(\texttt{x}, \texttt{DUMMY}), 2\rangle\!\rangle$. Similar to the case in list constructors, delayed initialization is expressed by application, namely $\langle\!\langle \texttt{append}(\texttt{x}, \texttt{DUMMY}), 2\rangle\!\rangle\ \texttt{y} = \texttt{append}(\texttt{x}, \texttt{y})$. Expressions of double angle brackets are executed and reduced to single angle brackets $\langle str, loc\rangle$, and application of `y` to this $\langle\ \rangle$-term will return the same result as $\texttt{append}(\texttt{x}, \texttt{y})$.

These $\langle\!\langle\ \rangle\!\rangle$-expressions work almost in the same way as constructors or constructing functions, except their result is a $\langle\ \rangle$-term. Note that $\langle\!\langle\ \rangle\!\rangle$-expressions are set always to return $\langle\ \rangle$-terms, even if abstracted context parameters disappear and are not reflected in the final results. In such cases loc of the resulting $\langle\ \rangle$-term is empty, and terms or expressions appearing on its right are just thrown away from the result. In order to denote this, we use -1 as the position number of $\langle\!\langle\ \rangle\!\rangle$-expressions.

4.3 Execution of the Extended Language

We first explain the basic concept of evaluation semantics which evaluates $\langle\!\langle\ \rangle\!\rangle$-expressions into $\langle\ \rangle$-terms. After that application rules to $\langle\ \rangle$-terms are introduced.

4.3.1 Evaluation Semantics

The basic idea of evaluating $\langle\!\langle\ \rangle\!\rangle$-expressions is that during execution, namely construction, we collect the information of the parent constructors which have `DUMMY` as a direct child and its position in Dewey notation. This information is returned as the location information which appears as the second element in the resulting $\langle\ \rangle$-term.

(1) Allocating a constructor over `DUMMY` makes a $\langle\ \rangle$-term, e.g.,

$$eval(\texttt{Cons}[\texttt{x1},\texttt{DUMMY}]) \Rightarrow \langle[\![\texttt{Cons}]\!][eval(\texttt{x1}),[\![\texttt{DUMMY}]\!]], loc\rangle,$$

where *loc* holds a tuple of the pointer to the allocated `Cons` cell and the position number 2.

(2) Allocating a constructor over $\langle\ \rangle$-terms again makes a $\langle\ \rangle$-term, e.g.,

$$eval(\texttt{Cons}[\texttt{x1},\langle str, loc\rangle]) \Rightarrow \langle[\![\texttt{Cons}]\!][eval(\texttt{x1}), str], loc\rangle.$$

In case there are plural $\langle\ \rangle$-terms in one constructor, these location informations point to the same abstracted value. Therefore both location informations are concatenated ($+\!\!+$) and returned as the new location information, e.g.,

$$eval(\texttt{Cons}[\langle strl, locl\rangle,\langle strr, locr\rangle]) \Rightarrow \langle[\![\texttt{Cons}]\!][strl, strr], locl +\!\!+ locr\rangle.$$

When returning a $\langle\ \rangle$-term as the final result, two special cares are needed:

- When only $[\![\texttt{DUMMY}]\!]$ is returned, the result of a $\langle\!\langle\ \rangle\!\rangle$-expression is $\langle\!\langle\texttt{DUMMY},\texttt{0}\rangle\!\rangle$.
- When the returned result is not a $\langle\ \rangle$-term but ordinary construction *str*, or when we evaluate $\langle\!\langle\ \rangle\!\rangle$-expression with its position number -1, it means that the abstracted parameter does not appear in that construction. Hence the final result is $\langle str, loc\rangle$ where *loc* is empty.

4.3.2 Application

With these allocated $\langle\ \rangle$-terms, we need another semantics of application. Now we use the assignment operation $\hookleftarrow$ shown in Section 4.1. Application of an expression exp_r to a $\langle\ \rangle$-term $aterm_l = \langle strl, locl\rangle$, expressed by $aterm_l\ exp_r$, follows the execution of $\hookleftarrow$:

- evaluate the right exp_r into $term_r$,
- assign the pointer to or the value of $term_r$ into the left structure *strl*, using $\hookleftarrow$ with the location information *locl*,
- return the pointer to the left structure *strl*.

In case the right expression is a $\langle\!\langle\ \rangle\!\rangle$-expression $aexp_r$, application returns again a $\langle\ \rangle$-term. In general they are evaluated as:

- evaluate the right $\langle\!\langle\ \rangle\!\rangle$-expression $aexp_r$ into a $\langle\ \rangle$-term $\langle strr, locr\rangle$;
- assign the pointer *strr* into the left structure *strl*, using $\hookleftarrow$ with the location information *locl*;
- return a new $\langle\ \rangle$-term $\langle strl, locr\rangle$.

In the previous subsection we used an example $eval(\texttt{Cons}[\texttt{x1},\langle str, loc\rangle])$ which results in $\langle[\![\texttt{Cons}]\!][eval(\texttt{x1}), str], loc\rangle$. This transformation is also the result of application. If we regard the second parameter from this `Cons` is abstracted out,

$$\begin{aligned}&eval(\texttt{Cons}[\texttt{x1},\langle str, loc\rangle])\\&= eval(\langle\!\langle\texttt{Cons}[\texttt{x1},\texttt{DUMMY}],2\rangle\!\rangle)\ \langle str, loc\rangle\\&= \langle[\![\texttt{Cons}]\!][eval(\texttt{x1}),[\![\texttt{DUMMY}]\!]], locl\rangle\langle str, loc\rangle\\&= \langle[\![\texttt{Cons}]\!][eval(\texttt{x1}), str], loc\rangle,\end{aligned}$$

where *locl* holds the pointer to the cell allocated by `Cons` and the position number 2.

We have to take care when the left $\langle\!\langle\ \rangle\!\rangle$-expressions or their evaluated $\langle\ \rangle$-terms has an empty location information in *loc*. When the right expressions are $\langle\!\langle\ \rangle\!\rangle$-expressions, the application returns again a $\langle\!\langle\ \rangle\!\rangle$-expressions, and the new location information holds not *locr* in the right expression but again an empty location information. This is because any expressions appearing on their right are thrown away but we have to keep a style of $\langle\ \rangle$-terms.

4.4 Obtaining Definitions for $\langle\!\langle\ \rangle\!\rangle$-functions

Evaluation of $\langle\!\langle\ \rangle\!\rangle$-expressions follows almost the same as standard semantics, except it collects the location information. This section investigates function definitions which has $\langle\!\langle\ \rangle\!\rangle$-expressions as their components. We call such expressions $\langle\!\langle\ \rangle\!\rangle$-functions.

We take an example of $\langle\!\langle\texttt{append(x,DUMMY)},2\rangle\!\rangle$. When $\texttt{x} = \texttt{Nil}$, it returns $\langle\!\langle\texttt{DUMMY},\texttt{0}\rangle\!\rangle$. If `x` can be decomposed into `Cons[x1, xs]`, the result will be $\langle\!\langle\texttt{Cons[x1,app(xs,DUMMY)]},22\rangle\!\rangle$. With the help of constructor abstraction this is the same as $\langle\!\langle\texttt{Cons[x1,DUMMY]},2\rangle\!\rangle\ \langle\!\langle\texttt{app(xs,DUMMY)},2\rangle\!\rangle$. Since the left $\langle\!\langle\ \rangle\!\rangle$-expression only contains constructors and we just have the definition of the right $\langle\!\langle\ \rangle\!\rangle$-function, transformation terminates. Replacing $\langle\!\langle\ \rangle\!\rangle$-function calls to `app2(x)`☆, we have the following definition:

```
define app2(x) case x of
  Nil → ⟨⟨DUMMY, 0⟩⟩
  Cons[x1, xs]
       → ⟨⟨Cons[x1, DUMMY]2⟩⟩ app2(xs).
```

As another example, we take `tflat(x, y)` which flattens a tree `x` into a list when `y` is `Nil`.

☆ New definition names takes the position number of the abstracted parameter as their suffix. We use `app2` instead of `append2` in this paper.

```
define tflat(x, y) case x of
  Leaf[n] → Cons[n, y]
  Node[l, r]
          → tflat(l, tflat(r, y))
```

This tflat is a functional constructor with respect to the second parameter y, and we have

$\texttt{tflat(x, y)} = \langle\!\langle \texttt{tflat(x, DUMMY)}, 2 \rangle\!\rangle\ \texttt{y}.$

The result of the abstracted functional constructor is ⟪Cons[n, DUMMY], 2⟫ in its Leaf branch; when x = Node[l, r], it will return tflat(l, tflat(r, DUMMY)). Since tflat is a functional constructor with respect to the second parameter, the inner tflat(r, DUMMY) goes out of the outer tflat call, and we have ⟪tflat(l, DUMMY), 2⟫ ⟪tflat(r, DUMMY), 2⟫. Replacing ⟪tflat(x, DUMMY), 2⟫ to tflat2(x) returns

```
define tflat2(x) case x of
  Leaf[n] → ⟪Cons[n, DUMMY], 2⟫
  Node[l, r]
          → tflat2(l) tflat2(r).
```

The restriction for obtaining definition of ⟪ ⟫-functions is that the context parameter to be abstracted appears at most one time in the following function call. For example, in the following definitions foo and bar

```
define foo(x, y)
   bar(x, y, y)

define bar(x, y, z) case x of
   Nil          → Cons[y, z]
   Cons[x1, xs] → bar(xs, Cons[x1, z], y),
```

⟪bar(x, DUMMY, z), 2⟫ and ⟪bar(x, y, DUMMY), 3⟫ can be defined as ⟪ ⟫-functions. However, ⟪foo(x, DUMMY), 2⟫ cannot have its definition as a ⟪ ⟫-function because the pointer to the abstracted value cannot be kept when we have to evaluate bar(xs, Cons[x1, DUMMY], DUMMY).

The transformation rules for obtaining the definition of ⟪ ⟫-functions are described in **Fig. 8**. As we can see, if each ⟪ ⟫-expression is not evaluated into a ⟨ ⟩-term, ⟪ ⟫-functions returns a sequence of ⟪ ⟫-expressions.

4.5 Properties of Abstracted Constructors

Now we investigate the properties of ⟪ ⟫-expressions or ⟪ ⟫-functions, and their evaluated results ⟨ ⟩-terms. We have already mentioned that ⟪ ⟫-expressions are adapted representations of lambda abstraction toward constructors. When considering application to ⟪ ⟫-expressions, they are first evaluated to ⟨ ⟩-terms and structures in the ⟨ ⟩-term are allocated in the heap; we then need assignments into the structures in the ⟨ ⟩-terms using the location information *loc*. While this evaluation is done in an interpretive manner, application itself does not cost much, since the needed operations are assignments into some known constructor cells and an assignment operation is a cheap operation in most of programming languages.

So far location information *loc* in ⟨ ⟩-terms is represented in the form of a list. In this naive representation, concatenation of two location informations $locl +\!\!+ locr$ needs computation proportional to the length of *locl*. Thus, for an efficient implementation, we propose to use cyclic lists[28]. Cyclic lists has a pointer pointing to the tail of the list, and the head and tail of the list is easily detected. This is similar to ⟨ ⟩-terms, and concatenation of two cyclic lists is done in constant time.

Indeed we utilize assignment operations, note that they are not really assignment operations, in the meaning to overwrite environments and cause side-effects. We use assignments for delaying their initialization, and no more update will take place, which is similar to static single assignment (SSA)[3),39)].

If we only care about DUMMY appearing in expressions, there is no need to show the position number in ⟪ ⟫-expressions like ⟪Cons[x1, DUMMY], 2⟫. It is possible to use this position number for verifying whether DUMMY appears in the correct position. In case the subexpression pointed by the position number is not DUMMY, evaluation of this ⟪ ⟫-expression should mean overwriting of the pointed subexpression. The system can detect this before evaluating the expression, and an error will be returned. When isolated DUMMY appears in ⟪ ⟫-expressions without pointed by the position number, such ⟪ ⟫-expressions creates holes which are never filled in the successive computation. The system again is possible to detect such errors beforehand. This gives us safety for programmers to use ⟪ ⟫-expressions.

Finally, the important property by the extension is associativity in applications. This is supported by the Church-Rosser property of lambda terms. For example, we assume an sequence of expressions $aexp_1\ aexp_2\ exp_3$ is given, where $aexp_1$ and $aexp_2$ are evaluated into ⟨ ⟩-terms $aterm_1$ and $aterm_2$, respectively, and exp_3 is evaluated into $term_3$. First, the evaluation order of each expressions does not matter to the final result, for our extended language assumes no side-effects. Second, the application

The transformation of a $\langle\!\langle\ \rangle\!\rangle$-function f starts from Es_n to eliminate the n-th parameter, and new $\langle\!\langle\ \rangle\!\rangle$-functions are given their name as f_n.

$$Es_n[\![\texttt{define } f(v_1 ... v_m)\ b]\!] \Rightarrow \texttt{define } f_n(v_1 ... v_{n-1}, v_{n+1} ... v_m)\ E[\![b]\!]\sigma[\mathit{vname} \mapsto v_n]$$

$$E[\![\texttt{case } t \texttt{ of } \{pat \to b\}+]\!]\sigma \Rightarrow \texttt{case } t \texttt{ of } \{pat \to E[\![b]\!]\sigma\}+$$

In case $\sigma[\mathit{vname}]$ does not appear in the defined body b, the following one rule apply:

$$E[\![b]\!]\sigma \Rightarrow \langle\!\langle b, -1 \rangle\!\rangle$$

In case $\sigma[\mathit{vname}]$ appears in the defined body, the following rules apply.
In each transformation, we assume e_i or v are $\sigma[\mathit{vname}]$ itself or contains $\sigma[\mathit{vname}]$:

$$E[\![c[e_1 ... e_m]]\!]\sigma \Rightarrow Ec[\![e_i]\!]\ \sigma \begin{bmatrix} \mathit{left} \mapsto c[e_1 .. e_{i-1}, \\ \mathit{right} \mapsto e_{i+1} ... e_m], \\ \mathit{pos} \mapsto i, \quad \mathit{out} \mapsto \epsilon \end{bmatrix}$$

$$E[\![f(e_1 ... e_m)]\!]\sigma \Rightarrow Ef[\![e_i]\!]\ \sigma \begin{bmatrix} \mathit{out} \mapsto f_i(e_1 ... e_{i-1}, e_{i+1} ... e_m)\ , \\ \mathit{left} \mapsto \epsilon, \quad \mathit{right} \mapsto \epsilon, \quad \mathit{pos} \mapsto \epsilon \end{bmatrix}$$

$$E[\![v]\!]\sigma \Rightarrow \langle\!\langle \texttt{DUMMY}, \texttt{0} \rangle\!\rangle$$

$$Ec[\![c[e_1 ... e_m]]\!]\sigma \Rightarrow Ec[\![e_i]\!]\ \sigma \begin{bmatrix} \mathit{left} \mapsto \sigma[\mathit{left}]\ c[e_1 ... e_{i-1}, \\ \mathit{right} \mapsto e_{i+1} ... e_m]\ \sigma[\mathit{right}], \\ \mathit{pos} \mapsto \sigma[\mathit{pos}]\ i \end{bmatrix}$$

$$Ec[\![f(e_1 ... e_m)]\!]\sigma \Rightarrow Ef[\![e_i]\!]\ \sigma \begin{bmatrix} \mathit{out} \mapsto \sigma[\mathit{out}]\ \langle\!\langle\ \sigma[\mathit{left}]\ \texttt{DUMMY}\ \sigma[\mathit{right}],\ \sigma[\mathit{pos}]\ \rangle\!\rangle \\ \quad f_i(e_1 ... e_{i-1}, e_{i+1} ... e_m) \\ \mathit{left} \mapsto \epsilon, \quad \mathit{right} \mapsto \epsilon, \quad \mathit{pos} \mapsto \epsilon \end{bmatrix}$$

$$Ec[\![v]\!]\sigma \Rightarrow \sigma[\mathit{out}]\ \langle\!\langle\ \sigma[\mathit{left}]\ \texttt{DUMMY}\ \sigma[\mathit{right}], \sigma[\mathit{pos}]\ \rangle\!\rangle$$

$$Ef[\![c[e_1 ... e_m]]\!]\sigma \Rightarrow Ec[\![e_i]\!]\sigma \begin{bmatrix} \mathit{left} \mapsto c[e_1 .. e_{i-1}, \\ \mathit{right} \mapsto e_{i+1} ... e_m], \\ \mathit{pos} \mapsto i \end{bmatrix}$$

$$Ef[\![f(e_1 ... e_m)]\!]\sigma \Rightarrow Ef[\![e_i]\!]\sigma \begin{bmatrix} \mathit{out} \mapsto \sigma[\mathit{out}]\ f_i(e_1 ... e_{i-1}, e_{i+1} ... e_m)\ , \\ \mathit{left} \mapsto \epsilon, \quad \mathit{right} \mapsto \epsilon, \quad \mathit{pos} \mapsto \epsilon \end{bmatrix}$$

$$Ef[\![v]\!]\sigma \Rightarrow \sigma[\mathit{out}]$$

- ϵ denotes an empty string.

Fig. 8 Translation rules for obtaining definition of $\langle\!\langle\ \rangle\!\rangle$-functions.

of the resulting terms can also start anywhere, and again, the order of application does not affect other parts of expressions or terms. This is because side-effecting assignments are enclosed in $\langle\ \rangle$-terms. To sum up,

$$\mathit{aterm}_{1,2}\ \mathit{term}_3 = \mathit{aterm}_1\ \mathit{term}_{2,3}$$

holds, where $\mathit{aterm}_{1,2}$ denotes the result of the application of aterm_1 to aterm_2, and $\mathit{term}_{2,3}$ denotes the result of the application of aterm_2 to term_3.

These properties enable fast execution by recursion removal with partial evaluation.

5. Recursion Removal

Now that we find associativity in constructors and functional constructors, we proceed to eliminate recursion from constructing functions. The idea follows what we have seen as transformation using lambda abstraction in Section 2.2.

The transformation rules are defined in **Fig. 9**. The detailed steps of transformation are described below.

Note that the transformed program can use the assignment operations $\hookleftarrow$. For readability we also use `let` expressions to bind local variables.

5.1 First Step: Recursion Removal in the Extended Language

The first step of transformation is the introduction of abstraction using $\langle\!\langle\ \rangle\!\rangle$-expressions and accumulation of these expressions into accumulating parameters.

5.1.1 Preprocessing

Before transformation, we need to check whether defined functions are functional con-

The transformation of recursion removal toward an ordinary function f starts from R, and new functions are given their name as f'.

$$R[\![\texttt{define}\ f(v_1...v_m)\ b]\!]$$
$$\Rightarrow \texttt{define}\ f'(v_1...v_m, \texttt{acc})\ R[\![b]\!]\sigma[out \mapsto \texttt{acc},\ nlist \mapsto [f]]$$

$$R[\![\texttt{case}\ t\ \texttt{of}\ \{pat \to b\}+]\!]\sigma$$
$$\Rightarrow \texttt{case}\ t\ \texttt{of}\ \{pat \to R[\![b]\!]\sigma\}+$$

$$R[\![v]\!]\sigma \Rightarrow \sigma[out]\ v$$

$$R[\![a]\!]\sigma \Rightarrow \sigma[out]\ a$$

$$R[\![f(e_1...e_m)]\!]\sigma \Rightarrow f'(T[\![e_1]\!]...T[\![e_m]\!], \sigma[out])$$

— there is no function call at all in e_j for $\forall j$ with flag *abst yes*, or there are no parameters in f with flag *abst yes*, or $f \in \sigma[nlist]$

$$\Rightarrow R[\![e_j]\!]\ \sigma \begin{bmatrix} out \mapsto \sigma[out] \\ \quad f'_j(\ T[\![e_1]\!]...T[\![e_{j-1}]\!], T[\![e_{j+1}]\!]...T[\![e_m]\!], \langle\!\langle \texttt{DUMMY}, 0\rangle\!\rangle, \langle\!\langle \texttt{DUMMY}, 0\rangle\!\rangle) \end{bmatrix}$$

— a function call whose flag *def* is *yes* exists in e_j

$$\Rightarrow A[\![e_j]\!]\ \sigma \begin{bmatrix} left \mapsto f(\ e_1...e_{j-1}, \\ pos \mapsto j, \\ right \mapsto e_{j+1}...e_m\ ) \end{bmatrix}$$

— a function call whose flag *abst* is *yes* and *def* is *no* exists in e_j

$$R[\![c[e_1...e_m]]\!]\sigma \Rightarrow \sigma[out]\ c[\ T[\![e_1]\!]...T[\![e_m]\!]\ ]$$

— there is no function call at all in e_j for $\forall j$

$$\Rightarrow A[\![e_j]\!]\ \sigma \begin{bmatrix} left \mapsto c[\ e_1...e_{j-1}, \\ pos \mapsto j, \\ right \mapsto e_{j+1}...e_m\ ] \end{bmatrix}$$

— a function call exists in e_j

$$A[\![f(e_1...e_m)]\!]\sigma \Rightarrow f'(T[\![e_1]\!]...T[\![e_m]\!], \sigma[out]\ \langle\!\langle \sigma[left]\ \texttt{DUMMY}\ \sigma[right], \sigma[pos]\rangle\!\rangle)$$

— there is no function call at all in e_j for $\forall j$ with flag *abst yes*, or there are no parameters in f with flag *abst yes*, or $f \in \sigma[nlist]$

$$\Rightarrow R[\![e_j]\!]\ \sigma \begin{bmatrix} out \mapsto \sigma[out]\ \langle\!\langle \sigma[left]\ \texttt{DUMMY}\ \sigma[right], \sigma[pos]\rangle\!\rangle \\ \quad f'_j(T[\![e_1]\!]...T[\![e_{j-1}]\!], T[\![e_{j+1}]\!]...T[\![e_m]\!], \langle\!\langle \texttt{DUMMY}, 0\rangle\!\rangle, \langle\!\langle \texttt{DUMMY}, 0\rangle\!\rangle), \\ left \mapsto \epsilon,\ pos \mapsto \epsilon,\ right \mapsto \epsilon \end{bmatrix}$$

— a function call whose flag *def* is *yes* exists in e_j

$$\Rightarrow A[\![e_j]\!]\ \sigma \begin{bmatrix} left \mapsto \sigma[left]\ f(\ e_1...e_{j-1}, \\ pos \mapsto \sigma[pos]\ j, \\ right \mapsto e_{j+1}...e_m\ )\ \sigma[right] \end{bmatrix}$$

— a function call whose flag *abst* is *yes* and *def* is *no* exists in e_j

$$A[\![c[e_1...e_m]]\!]\sigma \Rightarrow \sigma[out]\ c[\ T[\![e_1]\!]...T[\![e_m]\!]\ ]$$

— there is no function call at all in e_i for $\forall i$; this case will not happen

$$\Rightarrow A[\![e_j]\!]\ \sigma \begin{bmatrix} left \mapsto \sigma[left]\ c[\ e_1...e_{j-1}, \\ pos \mapsto \sigma[pos]\ j, \\ right \mapsto e_{j+1}...e_m\ ]\ \sigma[right] \end{bmatrix}$$

— a function call exists in e_j

The operation $e : s$ adds an element e to a set s.

Fig. 9 Transformation rules of recursion removal from functions (Part 1).

The transformation of recursion removal toward a $\langle\!\langle\ \rangle\!\rangle$-function f_n starts from R', and new functions are given their name as f'_n.
The definition of f_n is assumed to have been obtained already by Es_n, and R' applies over the definition.

$$R'[\![\texttt{define}\ f_n(v_1...v_{n-1}, v_{n+1}...v_m)\ b]\!] \Rightarrow \texttt{define}\ f'_n(v_1...v_{n-1}, v_{n+1}...v_m, \texttt{accl}, \texttt{accr})\ R'[\![b]\!]\ \sigma\,[nlist \mapsto [f_n]]$$

$$R'[\![\texttt{case}\ t\ \texttt{of}\ \{pat \to b\}+]\!]\sigma \Rightarrow \texttt{case}\ t\ \texttt{of}\ \{pat \to R'[\![b]\!]\sigma\}+$$

- In case the body b has only one $\langle\!\langle\ \rangle\!\rangle$-expression or $\langle\!\langle\ \rangle\!\rangle$-function call, the following rules apply:
$$R'[\![\langle\!\langle b, -1\rangle\!\rangle]\!]\sigma \Rightarrow \texttt{accl}\ \langle\!\langle b, -1\rangle\!\rangle\ \texttt{accr}$$
$$R'[\![\langle\!\langle c[e_1...e_m], j\rangle\!\rangle]\!]\sigma \Rightarrow \texttt{accl}\ \langle\!\langle c[e_1...e_m], j\rangle\!\rangle\ \texttt{accr}$$
$$R'[\![f_n(e_1...e_m)]\!]\sigma \Rightarrow f'_n(T[\![e_1]\!]...T[\![e_m]\!], \texttt{accl}, \texttt{accr})$$
- In case the body b has plural $\langle\!\langle\ \rangle\!\rangle$-expressions, there appears at least one $\langle\!\langle\ \rangle\!\rangle$-function call. If there is only one f_n, f_n is selected. When there are plural function calls, the leftmost one of original function calls $f_n = \sigma[nlist]$ is selected if it exists; one call out of them is selected if there is no $f_n = \sigma[nlist]$. For the selected f_n, the following rule applies, where *expsl* and *expsr* range over sequences of $\langle\!\langle\ \rangle\!\rangle$-expressions of length equal to or more than 0:
$$R'[\![expsl\ f_n(e_1...e_m)\ expsr]\!]\sigma \Rightarrow f'_n(T[\![e_1]\!]...T[\![e_m]\!], \texttt{accl}\ expsl, expsr\ \texttt{accr})$$

Following rules T apply to change function f into recursion removed function call f', which has the initial value $\langle\!\langle \texttt{DUMMY}, \texttt{0}\rangle\!\rangle$ in the accumulator `acc`.

$$T[\![v]\!] \Rightarrow v \qquad T[\![c[e_1...e_m]]\!] \Rightarrow c[T[\![e_1]\!]...T[\![e_m]\!]]$$
$$T[\![a]\!] \Rightarrow a \qquad T[\![f(e_1...e_m)]\!] \Rightarrow f'(T[\![e_1]\!]...T[\![e_m]\!], \langle\!\langle \texttt{DUMMY}, \texttt{0}\rangle\!\rangle)$$

Fig. 9 Transformation rules of recursion removal from functions (Part 2).

structors with respect to which parameters. In order to be a functional constructor, the parameter may not be tested by a `case` expression. The translation first needs to determine which functions can be functional constructors with respect to which parameters and whether the abstracted functional constructor can be defined as a $\langle\!\langle\ \rangle\!\rangle$-function. These are indicated by flags *abst* and *def*, respectively.

In the resulting table, *usage* shows whether the parameter is a recursion parameter (RP) or a context parameter (CP). *abst* shows whether the parameter can be abstracted out, and *def* shows whether the functional constructor can be defined as a $\langle\!\langle\ \rangle\!\rangle$-function with respect to the parameter. Note that *abst* is not always *yes* when the parameter is *CP*, because a context parameter in one function definition may be used as a recursion parameter in the following function calls. Additionally, when *def* is set to *yes*, its *abst* is also set to *yes*.

In the first pass, programs are simply manipulated textually. If a parameter is decomposed by `case` expressions, *abst* and *def* are set to *no*. Other parameters are context parameters in the function definition, and how they appear in the defined body are shown in *appearance* in the table. If the parameter is either directly output, which is denoted by `0`, or disappears in any branch, denoted by $-$`1`, then *abst* and *def* are set to *yes*; otherwise, they are left unflagged '-'. This pass gives a table shown in **Table 1**, except for leaving a hole '-' in *abst* and *def* of `append`'s second parameter `y`. This is because it is not yet clear whether the parameter appearing as a parameter in some other function is also possible to be abstracted, etc.

The flags left '-' in the table are filled by manipulating the table again in the second pass. This pass starts from any parameter with unflagged *abst* and checks whether there appear some parameters with *abst* set to *no*. When found, this means that the parameter has a possibility later to be used as a recursion parameter, and it immediately returns *no* for *abst*. When it reaches back to the same parameter in the same function or reaches *yes* in every possible branching, then the original variable is safely set to be *yes* and table is completed. In this process *def* is also investigated in the same manner.

As the result of table making, we obtain the full contents in Table 1. This table shows that `append` is categorized as a functional construc-

Table 1 Result of table making. Numbers appearing before colon show the branching by conditions, following Dewey notation. *RP* means a recursion parameter, *CP* a context parameter. Flag *abst* shows whether the function can be functional constructor with respect to the parameter. Flag *def* shows whether the functional constructor can be defined as ⟪ ⟫-function.

funct	*var*	*usage*	*abst*	*def*	*appearance*
`append`	x	*RP*	*no*	*no*	
	y	*CP*	*yes*	*yes*	1: 0
					2: $2_{\texttt{Cons}}2_{\texttt{append}}$
`lflat`	x	*RP*	*no*	*no*	
`flip`	x	*RP*	*no*	*no*	

tor with respect to the second parameter `y`, and it can be defined as a ⟪ ⟫-function.

5.1.2 Transformation

The translation proceeds with referring to the table. It translates programs written in the source language in Fig. 2 into their recursion removed form written in the extended language.

The transformation rules R apply to function definitions, which gives a new function name and a new parameter `acc` for accumulation. In each branch one function, if it exists, is selected and changed to its recursion removed call. The remaining part is expressed as ⟪ ⟫-expressions and such ⟪ ⟫-expressions are accumulated by applying it on the right of `acc` in the selected function call. If there is no function calls, the inherited result in `acc` is returned with the branch body.

⟪`DUMMY`, `0`⟫ works as the unit term. New function calls therefore takes ⟪`DUMMY`, `0`⟫ in its accumulator.

One important decision is how deep we go into. There are cases where naive transformation can worsen stack usage.

Contrasting Examples: `reverse` and `rflat`. `reverse` takes a list and returns a reversed list; `rflat` takes a list of lists and returns a reversed flattened list.

```
define reverse(x, y) case x of
 Nil          → y
 Cons[x1, xs] → reverse(xs,
                  Cons[x1, y])

define rflat(x, y) case x of
 Nil          → y
 Cons[x1, xs] → rflat(xs,
                  reverse(x1, y))
```

Stack usage of both functions are bounded (max is two). Since `rflat` as well as `reverse` is a functional constructor with respect to the second parameter `y`, the system may separate the `Cons[x1, xs]` branch of `rflat` into ⟪`rflat(xs, DUMMY), 2`⟫ `reverse(x1, y)` and try to accumulate ⟪`rflat(xs, DUMMY), 2`⟫ inside of the renamed call `reverse'(x1, y, acc)`. Evaluation of ⟪`rflat(xs, DUMMY), 2`⟫ makes an independent stack frame for `reverse'` recursively and worsens stack usage.

In order to prevent such commission errors, the separation stops by the direct call to itself.

Recursion removal is also possible from ⟪ ⟫-functions. Transformation rule R' applies to definitions of ⟪ ⟫-functions, and accumulates ⟪ ⟫-expressions into a recursive ⟪ ⟫-function call if it exists. For example, `app2(x)` which is the function definition of ⟪`append(x, DUMMY), 2`⟫ will return ⟪`Cons[x1, DUMMY], 2`⟫ `app2(xs)` in the `Cons` branch. The abstracted ⟪`Cons[x1, DUMMY], 2`⟫ is accumulated into the recursive call. We put the restriction that the separation will not go into its direct call. For ⟪ ⟫-functions, outer function calls or constructors appear on the left, and inner ones appear on its right. We here put the same restriction that selection will stop before its direct call on its leftmost position, and the remaining ⟪ ⟫-expressions are accumulated inside of the recursive call.

There is one difference on accumulation from ordinary functions. That is, ⟪ ⟫-functions accumulate not only the ⟪ ⟫-expressions on the left of a recursive call, but also ⟪ ⟫-expressions on the right.

Contrasting Example: `mir`. `mir` takes a list and returns its mirrored list:

```
define mir(x, y) case x of
 Nil          → y
 Cons[x1, xs] → Cons[x1, mir(xs,
                  Cons[x1, y])].
```

`mir` is a ⟪ ⟫-function with respect to the second parameter, and its definition `mir2(x)` is given as follows:

```
define mir2(x) case x of
  Nil          → ⟪DUMMY, 0⟫
  Cons[x1, xs] → ⟪Cons[x1, DUMMY], 2⟫
                 mir2(xs) ⟪Cons[x1, DUMMY], 2⟫.
```

Cons branch has ⟪Cons[x1, DUMMY], 2⟫ on each side of the recursive call mir2(xs). When we apply recursion removal on mir2(x), both outputs are accumulated inside of the recursive call. We therefore prepare two accumulators accl and accr, and they accumulate outputs on the left and right of the recursive call, respectively. While accl accumulates the output on its right, accr on the contrary accumulates the output on its left. Therefore the recursion removed function mir2' is defined as:

```
define mir2'(x, accl, accr) case x of
  Nil          → accl ⟪DUMMY, 0⟫ accr
  Cons[x1, xs] →
 mir2'(xs, accl ⟪Cons[x1, DUMMY], 2⟫,
           ⟪Cons[x1, DUMMY], 2⟫ accr).
```

The transformations R or R' create a new function definition f' from f or f'_j from f_j. Inside of the newly obtained definitions, the selected recursive call as well as other function calls which are not in ⟪ ⟫-expressions are renamed from g to g' with the initial value ⟪DUMMY, 0⟫. When such calls are not yet defined, the translation continues until all function calls are defined.

5.2 Second Step: Optimization by Specialization

The resulting definitions may not run fast since interpretive overhead for execution and application of ⟪ ⟫-expressions exist. In order to eliminate this overhead, we can apply the partial evaluation techniques.

5.2.1 Specialization on Parameters

Partial evaluation is a program transformation which partially evaluate programs using information of known inputs or program structures[26),27),37)]. We view the operational semantics in Fig. 7 as an interpreter and specialize it with respect to programs written in the extended language.

The question about the minimum specialization power required to specialize the semantics is left for future works. Here we have in mind is an online specialization technique. Regardless of the specialization techniques, the assignment operation $\hookleftarrow$ in Fig. 7 is always dynamic and will remain in the residual program. We will see this in our examples in Figs. 10 and 11.

For specialization by partial evaluation, the important point is whether the resulting location information of evaluated ⟪ ⟫-expressions is known at compile-time. Though the pointer to the parent constructors of abstracted value depends on the execution, the position of abstracted value in such constructors can be obtained beforehand. We take an example of ⟪Cons[x1, DUMMY], 2⟫. It is evaluated to a $\langle str, loc\rangle$ whose number of hole is always one and its position number in loc is always 2. This means that the assignment operation we need is always $\hookleftarrow_2$.

This information is easily obtained when the ⟪ ⟫-expressions in question have no function calls in the path toward DUMMY. When such ⟪ ⟫-expressions are applied to the accumulator, the next function call can be specialized for an assignment operation suitable for the new ⟪ ⟫-expression.

⟪DUMMY, 0⟫ in the accumulator is also optimized. This occurs when a new function is called without previous output. In the specialized function definitions interpretive overhead on the application to ⟪DUMMY, 0⟫ is eliminated beforehand.

Generally ⟪ ⟫-expressions including abstracted functional constructors are hard to predict the resulting location information. This is because even the number of holes in the concrete structure varies depending on the recursion parameters in the functional constructors. For some functions, however, the information is possible to analyze. What helps this is ⟪ ⟫-functions in Section 4. The point for defining ⟪ ⟫-functions is whether abstracted values appear at most once in its recursive calls. This property makes the analysis simpler.

What we need to know is, (1) when we evaluate the ⟪ ⟫-functions into a sequence of ⟪ ⟫-expressions, what is the rightmost ⟪ ⟫-expression and what assignment operation is needed, and (2) whether there appears $\langle\!\langle exp, -1\rangle\!\rangle$ in the sequence. $\langle\!\langle exp, -1\rangle\!\rangle$ appears when its abstracted parameter disappears from evaluation. Any other expressions on its right are not reflected in the result. This means that, when there is at least one $\langle\!\langle exp, -1\rangle\!\rangle$ in the sequence, we do not need to know what is the rightmost ⟪ ⟫-expression, and the other expressions on its right are evaluated but not reflected.

app2(x), for example, is a ⟪ ⟫-function of ⟪append(x, DUMMY), 2⟫. It returns ⟪DUMMY, 0⟫ when the recursive parameter is Nil, and in its Cons branch it returns in its recursive definition ⟪Cons[x1, DUMMY], 2⟫ app2(x). As we see, there

Source program:

```
define flip(x) case x of
  Leaf[n]     → Leaf[n]
  Node[l, r]  → Node[flip(r), flip(l)]
```

Transformation steps:

```
R⟦define flip(x) case x of                  define flip'(x, acc) case x of
   Leaf[n]    → Leaf[n]               ⇒        Leaf[n]    → R⟦Leaf[n]⟧σ
   Node[l, r] → Node[flip(r), flip(l)]⟧        Node[l, r] → R⟦Node[flip(r), flip(l)]⟧σ
                                                 where σ = [out ↦ acc, nlist ↦ [flip]]
```

- for `Leaf[n]` branch:

```
R⟦Leaf[n]⟧σ ⇒ acc Leaf[T⟦n⟧]
            ⇒ acc Leaf[n]
```

- for `Node[l, r]` branch:

```
R⟦Node[flip(r), flip(l)]⟧σ
⇒ A⟦flip(l)⟧σ [ left ↦ Node[flip(r), ] ,
                pos ↦ 2,  right ↦ ]
⇒ flip'(T⟦l⟧, acc ⟪Node[flip(r), DUMMY], 2⟫)
⇒ flip'(l, acc ⟪Node[flip(r), DUMMY], 2⟫)
```

Transformation result:

```
define flip'(x, acc) case x of
  Leaf[n]    → acc Leaf[n]
  Node[l, r] → flip'(l, acc ⟪Node[flip(r), DUMMY], 2⟫)
```

Specialization result (with proper renaming of function calls):

```
define flip'-init(x) case x of
  Leaf[n]    → Leaf[n]
  Node[l, r] →
    let head = Node[flip'-init(r), DUMMY]
        tail = head
     in flip'-2(l, head, tail)
```

```
define flip'-2(x, head, tail) case x of
  Leaf[n]    →
    let tmp = Leaf[n]
        tail = tail ↩₂ tmp
      in head
  Node[l, r] →
    let tmp = Node[flip'-init(r), DUMMY]
        tmpt = tmp
        tail = tail ↩₂ tmp
      in flip'-2(l, head, tmpt)
```

Fig. 10 Complete transformation of `flip`.

appears no $\langle\!\langle exp, -1\rangle\!\rangle$ in the resulting sequence. Once `x` matches to `Cons`, the position number is always 2 because the result in the terminating condition is the identity $\langle\!\langle$`DUMMY, 0`$\rangle\!\rangle$, which is eliminated, and $\langle\!\langle$`Cons[x1, DUMMY], 2`$\rangle\!\rangle$ on its left matters for later assignment. In case initially `x` equals to `Nil`, the result is $\langle\!\langle$`DUMMY, 0`$\rangle\!\rangle$ and it can be specialized out.

5.2.2 Replacement of Function Calls in $\langle\!\langle\ \rangle\!\rangle$-expressions

In the transformation R and R', function calls in $\langle\!\langle\ \rangle\!\rangle$-expressions are left as it is. This is because of the semantics of the extended language. However, when a function call does not have `DUMMY` as an subexpression, the function always returns a concrete structure without holes. The specializer takes care of this when specializing a program in the extended language, and it applies recursion removed function calls for such occurrences.

6. Two Complete Transformations

In this section we show the recursion removal from `flip` and `lflat`. Their transformation is summarized in **Figs. 10** and **11**. We now describe the transformation, especially specialization in the second step, in more detail.

6.1 Example 1: flip

`flip(t)` flips every node in the given tree. This is a tree recursion and `flip(l)` out of two recursive calls is selected. Figure 10 shows the process and result of transformation.

By the first step, the program is transformed into

Source programs:

```
define append(x, y) case x of
 Nil          → y
 Cons[x1, xs] → Cons[x1, append(xs, y)]

define lflat(x) case x of
 Nil          → Nil
 Cons[x1, xs] → append(x1, lflat(xs))
```

Transformation steps:

```
R⟦define lflat(x) case x of                   ⇒   define lflat'(x, acc) case x of
   Nil          → Nil                                 Nil          → R⟦Nil⟧σ
   Cons[x1, xs] → append(x1, lflat(xs))⟧              Cons[x1, xs] →
                                                              R⟦append(x1, lflat(xs))⟧σ
                                                  where σ = [out ↦ acc, nlist ↦ [lflat]]
```

- for Nil branch: $R[\![\texttt{Nil}]\!]\sigma \Rightarrow$ `acc Nil`
- for `Cons[x1, xs]` branch:

```
R⟦append(x1, lflat(xs))⟧σ ⇒ R⟦lflat(xs)⟧σ [ out ↦ σ[out] app2'(T⟦x1⟧,
                                                   ⟨⟨DUMMY, 0⟩⟩, ⟨⟨DUMMY, 0⟩⟩) ]
                          ⇒ lflat'(T⟦xs⟧, acc app2'(x1, ⟨⟨DUMMY, 0⟩⟩, ⟨⟨DUMMY, 0⟩⟩))
                          ⇒ lflat'(xs, acc app2'(x1, ⟨⟨DUMMY, 0⟩⟩, ⟨⟨DUMMY, 0⟩⟩))
```

```
Es2⟦define append(x, y) case x of             ⇒   define app2(x) case x of
   Nil          → y                                   Nil          → ⟨⟨DUMMY, 0⟩⟩
   Cons[x1, xs] → Cons[x1, append(xs, y)]⟧            Cons[x1, xs] → ⟨⟨Cons[x1, DUMMY], 2⟩⟩ app2(xs)

R′⟦define app2(x, y) case x of                ⇒   define app2'(x, accl, accr) case x of
   Nil          → ⟨⟨DUMMY, 0⟩⟩                        Nil          → accl ⟨⟨DUMMY, 0⟩⟩ accr
   Cons[x1, xs] → ⟨⟨Cons[x1, DUMMY], 0⟩⟩              Cons[x1, xs] → app2'(xs, accl
                     app2(xs)⟧                                     ⟨⟨Cons[x1, DUMMY], 2⟩⟩, accr)
```

Transformation results:

```
define lflat'(x, acc) case x of
  Nil          → acc Nil
  Cons[x1, xs] → lflat'(xs, acc
     app2'(x1, ⟨⟨DUMMY, 0⟩⟩, ⟨⟨DUMMY, 0⟩⟩))

define app2'(x, accl, accr) case x of
  Nil          → accl ⟨⟨DUMMY, 0⟩⟩ accr
  Cons[x1, xs] → app2'(xs, accl
                    ⟨⟨Cons[x1, DUMMY], 2⟩⟩, accr)
```

Specialization result (with proper renaming of function calls):

```
define app2'-init(x) case x of
  Nil          → ⟨⟨DUMMY, 0⟩⟩
  Cons[x1, xs] →
    let head = Cons[x1, DUMMY]
        tail = head
     in app2'-2(xs, head, tail)

define app2'-2(x, head, tail) case x of
  Nil          → ⟨head, [tail, 2]⟩
  Cons[x1, xs] →
    let tmp = Cons[x1, DUMMY]
        tmpt = tmp
        tail = tail ↩2 tmp
     in app2'-2(xs, head, tmpt)

define lflat'-init(x) case x of
  Nil          → Nil
  Cons[x1, xs] →
    case x1 of
      Nil → lflat'-init(xs)
      Cons[x11, x1s] →
        let ⟨head, [tail, 2]⟩
              = app2'-init(x1)
         in lflat'-2(xs, head, tail)

define lflat'-2(x, head, tail) case x of
  Nil          →   let tmp = Nil
                       tail = tail ↩2 tmp
                    in head
  Cons[x1, xs] →
    case x1 of
      Nil → lflat'-2(xs, head, tail)
      Cons[x11, x1s] →
        let ⟨tmp, [tmpt, 2]⟩
              = app2'-init(x1)
            tail = tail ↩2 tmp
         in lflat'-2(xs, head, tmpt)
```

Fig. 11 Complete transformation of `lflat` with `app2`.

```
define flip'(x,acc) case t of
  Leaf[n]    → acc Leaf[n]
  Node[l,r]  → flip'(l,acc
          ⟨⟨Node[flip(r),DUMMY],2⟩⟩).
```

As we have already mentioned, this definition includes interpretive overhead. First, the initial call for `flip'` is called with ⟨⟨`DUMMY,0`⟩⟩ in its accumulator `acc`. This is soon eliminated by partial evaluation, and we have

```
define flip'(x,⟨⟨DUMMY,0⟩⟩) case t of
  Leaf[n]    → Leaf[n]
  Node[l,r]  → flip'(l,
          ⟨⟨Node[flip(r),DUMMY],2⟩⟩).
```

The next function call to specialize is `flip'(l,⟨⟨Node[flip(r),DUMMY],2⟩⟩)` which appears in the `Node` branch. `flip(r)` does not have `DUMMY` as its subexpression, then it is safely replaced to `flip'(r,⟨⟨DUMMY,0⟩⟩)`. The `Node` branch can now be regarded as:

```
let head = Node[flip'(r,⟨⟨DUMMY,0⟩⟩),
                               DUMMY]
    tail = head
  in flip'(l,⟨head,[tail,2]⟩).
```

We then proceed to see the result of `flip'(l,⟨head,[tail,2]⟩)`:

```
define flip'(x,⟨head,[tail,2]⟩) case t of
  Leaf[n]    → ⟨head,[tail,2]⟩ Leaf[n]
  Node[l,r]  → ⟨head,[tail,2]⟩ flip'(l,
  ⟨⟨Node[flip'(l,⟨⟨DUMMY,0⟩⟩),DUMMY],2⟩⟩).
```

The `Leaf` branch is equivalent to:

```
let tmp = Leaf[n]
    tail = tail ↩₂ tmp
  in head,
```

and the `Node` branch is equivalent to:

```
let tmp = Node[flip'(r,⟨⟨DUMMY,0⟩⟩),
                               DUMMY]
    tmpt = tmp
    tail = tail ↩₂ tmp
  in flip'(l,⟨head,[tmpt,2]⟩).
```

The `Node` branch of `flip'(x,⟨head,[tail,2]⟩)` again calls `flip'(l,⟨head,[tmpt,2]⟩)`, and specialization is no more needed. We give proper names `flip'-init` and `flip'-2` for each function calls, and we have the final result as in Fig. 10.

6.2 Example 2: lflat

`lflat(x,y)` flattens a list of lists and accumulate the result in a parameter. Figure 11 shows the transformation and the result.

The first step separates ⟨⟨`append(x1,DUMMY)`, 2⟩⟩ whose function name is `app2` and accumulate it in a recursive call of `lflat`. `app2` is possible to recursion remove, and the new definition is

```
define lflat'(x,acc) case x of
  Nil          → acc Nil
  Cons[x1,xs]  → lflat'(xs,acc
     app2'(x1,⟨⟨DUMMY,0⟩⟩,⟨⟨DUMMY,0⟩⟩).
```

By giving ⟨⟨`DUMMY,0`⟩⟩ to `acc` and specialization, we have the initial call of `lflat`:

```
define lflat'(x,⟨⟨DUMMY,0⟩⟩) case x of
  Nil          → Nil
  Cons[x1,xs]  → lflat'(xs,
       app2'(x1,⟨⟨DUMMY,0⟩⟩,⟨⟨DUMMY,0⟩⟩).
```

We need to know about `app2'` for `lflat'` to be specialized. The definition of `app2'` is given as:

```
define app2'(x,accl,accr)
 case x of
  Nil          → accl ⟨⟨DUMMY,0⟩⟩ accr
  Cons[x1,xs]  → app2'(xs,
       accl ⟨⟨Cons[x1,DUMMY],2⟩⟩,accr).
```

Giving ⟨⟨`DUMMY,0`⟩⟩ to both `accl` and `accr`, we have the definition for initial calls:

```
define app2'(x,⟨⟨DUMMY,0⟩⟩,⟨⟨DUMMY,0⟩⟩)
 case x of
  Nil          → ⟨⟨DUMMY,0⟩⟩
  Cons[x1,xs]  → app2'(xs,
        ⟨⟨Cons[x1,DUMMY],2⟩⟩,⟨⟨DUMMY,0⟩⟩).
```

Similar to the case of `flip` in Section 6.1, its `Cons` branch is written using `let` and ↩:

```
let head = Cons[x1,DUMMY]
    tail = head
  in app2'(xs,⟨head,[tail,2]⟩,⟨⟨DUMMY,0⟩⟩).
```

`app2'(x,⟨head,[tail,2]⟩,⟨⟨DUMMY,0⟩⟩)` is specialized and written with `let` and ↩:

```
define app2'(x,⟨head,[tail,2]⟩,
                      ⟨⟨DUMMY,0⟩⟩)
 case x of
  Nil          → ⟨head,[tail,2]⟩
  Cons[x1,xs]  →
     let tmp = Cons[x1,DUMMY]
         tmpt = tmp
         tail = tail ↩₂ tmp
      in app2'(xs,⟨head,[tmpt,2]⟩,
                      ⟨⟨DUMMY,0⟩⟩).
```

We give names `app2'-init` and `app2'-2` to these calls, and specialization finishes.

As we see, `app2'` returns ⟨⟨`DUMMY,0`⟩⟩ when `x = Nil`, and ⟨*head*, [*tail*, 2]⟩ otherwise. This information is utilized for specialization of `lflat'`. There, one test for `x1` is sufficient and we have a specialized definition:

Table 2 Execution examples.

	total execution (gc)			
	recur.(sec)	iter.(sec)	iter./recur.	notes
`append`	31.46 (18.81)	3.53 (0.54)	0.112	`append(append(u, v), w)` for three lists `u`, `v` and `w` of length 30,000 for each, 100 times
`lflat`	4.96 (1.29)	2.10 (0.45)	0.42	`lflat(x)` for a list of length 1000 of lists of length 50, 100 times
`mergesort`	142.87 (58.96)	62.90 (21.33)	0.440	bottom-up mergesort for an uniform random sequence of length 60,000, 100 times
`flip`	3.67 (1.20)	3.87 (0.32)	1.05	`flip(t)` for an even binary tree of 2^{16} leaves, 100 times

```
define lflat'(x, ⟪DUMMY, 0⟫)
  case x of
    Nil          → Nil
    Cons[x1, xs] →
      case x1 of
        Nil → lflat'(xs, ⟪DUMMY, 0⟫)
        Cons[x11, x1s] →
            let aterm = app2'-init(x1)
             in lflat'(xs, aterm).
```

The next to investigate is `lflat'(xs,` *aterm*`)` in which *aterm* equals to $\langle head, [tail, 2]\rangle$:

```
define lflat'(x, ⟨head, [tail, 2]⟩)
 case x of
  Nil →  let tmp = Nil
             tail = tail ↩₂ tmp
         in head
  Cons[x1, xs] →
    case x1 of
     Nil → lflat'(xs, ⟨head, [tmpt, 2]⟩)
     Cons[x11, x1s] →
      let ⟨tmp, [tmpt, 2]⟩
             = app2'-init(x1)
          tail = tail ↩₂ tmp
       in lflat'(xs, ⟨head, [tmpt, 2]⟩).
```

Now that all function calls are specialized, the transformation finishes. We can again give names `lflat'-init` and `lflat'-2` for each definition.

7. Experiments

We now examine the optimization achieved by our transformation using our three examples (`append`, `flip` and `lflat`) and `mergesort`. The experiments were performed on a Sun Ultra Enterprise 2 with 200 MHz dual UltraSparcI and SunOS 5.5.1, and Allegro Common Lisp 4.3.1. We compiled the programs with optimization settings of safety 1, space 1, speed 1 and debug 2. In this settings tail call optimization is done.

In our experiments, `mergesort` computes the result in a bottom-up manner, continuously merging neighboring two lists into one. This `mergesort` consists of three linear subroutines, and all three functions are recursion removed.

In order to run the assignment operations we need to choose an implementation of $\hookleftarrow$. In our example only $\hookleftarrow_2$ appears, and we implement it by `rplacd`. We also used `Nil` for the occurrences of `DUMMY`.

As **Table 2** shows, huge improvements are achieved in three cases (`append`, `lflat` and `mergesort`). These linear recursion improves by recursion removal about 2 to 10 times faster.

One surprising and disappointing result for us is the example of tree recursion `flip`. Execution time becomes a little worsened about 5%. Since there is no difference in transformation between linear and tree recursion, some optimization is originally done by the compiler for them.

Note that reduction of stacks has good effects on garbage collection. Since the garbage collector has to manipulate call stacks, the number of stack frames greatly matters for garbage collection[15]. Even though the number of allocated cells does not differ, time for garbage collection is reduced by recursion removal.

8. Other Applications

The main objective of this paper is to realize recursion removal for constructing functions. The idea of abstraction from constructors and functional constructors is not limited to recursion removal. This section gives a brief overview of its usability for other purposes.

8.1 Recursion Introduction

We have introduced $\langle\!\langle\ \rangle\!\rangle$-functions and their definition is obtained by the rules in Fig. 8. This transformation eliminates a context parameter and gives us a definition which consist of a sequence of $\langle\!\langle\ \rangle\!\rangle$-expressions. Its evaluation is done in an interpretive manner.

If we apply the rule Es_2 to `reverse` which appeared in Section 5.1, the resulting definition is

```
define reverse2(x) case x of
  Nil          → ⟪DUMMY, 0⟫
  Cons[x1, xs] → reverse2(xs)
                    ⟪Cons[x1, DUMMY], 2⟫.
```

Onc call of `reverse2` puts ⟪`Cons[x1, DUMMY], 2`⟫ on the right of the recursive call when the given recursion parameter is not `Nil`. This transformation makes expressions appearing over a context parameter explicit.

Such transformations have a large importance, though currently it is not stressed. There are several partial evaluation methods[21),22),41),44),45)], and there are cases that they works easily and terminates successfully in recursive definitions. Definitions using accumulating parameters actually suffer from nontermination or failure of partial evaluation, while recursive variants succeed. GPC, generalized partial computation, is one system of partial evaluation which utilizes theorem proving, and it is now in the stage of experimental implementation[20)]. GPC successfully composes `reverse` and `tflat` to produce a new definition which eliminates intermediate data, provided they are defined using `append`. In case they are defined in the accumulation style, transformation fails[29)].

8.2 Tupling

As one of the costs of recursive programming, there occurs repetition of the same, therefore redundant computation. Component-based programming also incur inefficiency to traverse the same input repeatedly. Such inefficiency is sometimes reduced or avoided by tupling method[37)].

For enabling tupling method, lambda abstraction works quite successfully[36)]. One example which is often used is `repmin(t)` = `rep(t,min(t))`, which finds the minimum in the tree and makes a new tree with the same structure, except every leaf has the minimum value. In call-by-value semantics, first `min` traverses the input tree `t` to find the minimum, and again `rep` traverses `t` to make a new tree. Since `rep` is a functional constructor with respect to the second input, we have `repmin(t)` = ⟪`rep(t, DUMMY), 2`⟫ `min(t)`.

By this separation tupling becomes quite simple because ⟪`rep(t, DUMMY), 2`⟫ and `min(t)` both traverses over the same structure. We here just leave the detail here.

8.3 Parallel Execution

⟪ ⟫-expressions and ⟪ ⟫-functions are separately executed and do not affect each other. This guarantees parallel execution of each separated ⟪ ⟫-expressions and ⟪ ⟫-functions. Each ⟪ ⟫-expressions and ⟪ ⟫-functions are evaluated into ⟨ ⟩-terms and later composed by application of ⟨ ⟩-terms.

9. Related Works

In this section we give a brief overview and comparison of the related works.

9.1 Data Structures

First, we compare with the previous works on delaying initialization of contents from structures. Historically the idea of these half-way construction or delaying initialization used in this paper has been of ordinary use in logic programming[42)]. In functional styles, however, such ideas are latecomers. `append` is interpreted as representation function[25)], and utilized to produce structures like difference lists. There, the help of pregiven associativity of `append` enables transformations to reduce complexity. Later I-structures is invented for efficient execution in parallel programming[7)]. I-structure is ‘a special kind of array, each of whose components may be written no more than once.’ Since I-structure is intended for parallel execution, especially for vector computation, this idea has a basis on arrays, not constructors. While array-based I-structure has its advantage over indexing, constructor-based ⟪ ⟫-expressions may take advantage of representing unbound size of construction, as is the case in lists.

9.2 Use of Lambda Abstraction

Based on lambda abstraction, the idea of hole abstraction has been proposed recently for recursion removal[32)]. Our idea can be seen as an extension to the hole abstraction. The first point is how to obtain definitions of ⟪ ⟫-functions. Though abstraction from functions also appears in that paper, transformation methods to obtain the definition are not described obviously. Our idea of abstraction from functional constructors enables us to achieve recursion removal from `lflat` without knowing associativity in `append`. Second, hole abstraction limits the number of holes in a concrete structure to one. It is true single holes are easy to analyze, but our natural idea gives more generality, and it is suitable not only for recursion removal, but also parallel execution same as I-structures. Third and the last point is, though our idea connects to more low-level execution like destructive assignments by `rplacd` for example, our representation enables faster execution as is demonstrated in Section 7.

The idea of lambda abstraction is often used in program transformation. Due to Church-Rosser property, results can be accumulated without explicitly investigating associativity of auxiliary functions[12]. Higher-order expressions are used to derive efficient programs by partial evaluation[36),38),47]. In our research, closure-like expressions of constructors are reduced into assignments by specialization.

9.3 Other Works on Recursion and Iteration

As is briefly mentioned in Section 2, the topic of recursion removal has been energetically researched for many years by several approaches. These techniques are described in two monographs[10),34]. One method, outside-in transformation, does not require explicit use of stacks, but information of associativity has to be given from outside. Recursion removal has been described as translation into flowcharts[8),43),46] using *schematology*[35], translation schemes using pattern matchings of program structures and function properties[18]. *Fold-unfold*[6),14] steps are later used to derive iterative solutions.

In the inside-out transformation, increment of programs are investigated[24),31),33]. Schematology also applies to this style of recursion removal[13].

To tackle with recursion removal for constructing functions, recently two ideas have appeared. One approach[30] is a inside-out manner, and since there is no inverse of `cdr` they manipulate input lists following Deutsch, Schorr, Waite algorithm[40] using destructive operations. This eliminates stacks of input chains. Our idea does not destroy input lists, and new constructions are kept in its halfway. The other[23] tackles with construction problems using pseudo-associativity, namely in a outside-in manner. Our idea in this paper extends this, and investigation of pseudo-associativity is eliminated by fixing the scope only to constructors.

Recursion removal is sometimes compared with continuation passing style (CPS)[4),19]. While CPS only collects the history of calculation, our method selects one function call and accumulation is passed only to the call, with leaving other calls almost intact, and execution are done at each steps of function calls.

Finally we make a short note that, compared with recursion removal, a term 'recursion introduction'[11] appears quite fewer as far as we have searched.

10. Conclusion

We presented a method for recursion removal which works in two steps. We first extended the language to have abstraction mechanism in the form of ⟪ ⟫-expressions. The abstraction enabled us to have associativity in constructors which originally they do not have. The first step of transformation accumulates these abstracted expressions inside of a recursive call, and gives us new definitions of recursion removed functions in the extended language. The second step specializes the new definition to embed the language extension, and fast execution using assignment operations is realized. This transformation is applicable not only to linear recursion but also to tree recursion, or even to certain forms of nested functions.

Currently the detailed analysis on the partial evaluator which is required for the second step remains for future works. As Section 6 demonstrated, its specialization is not so hard when `DUMMY` does not appear in functions inside of the accumulated ⟪ ⟫-expression. In case `DUMMY` appears in such functions, or ⟪ ⟫-functions are accumulated, the analysis becomes hard. We showed some ideas, but further research is required for automation.

In this paper we presented a theoretical study of a novel method for recursion removal based on abstraction from constructors and partial evaluation. Our next task is to test these ideas in an implementation. We expect this will be straightforward using transformation and semantics rules which are presented in this paper.

Another task is to investigate more detailed application of our idea of abstraction, especially for other area of partial evaluation. When a function is definable as a ⟪ ⟫-function, its program structures are decomposed into ⟪ ⟫-expressions. This makes program analysis on context parameters easier and will pave the way to more partial evaluation like functional composition.

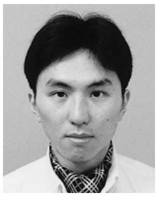
References

1) *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, USA, Jan. 1988, ACM Press (1988).
2) Aho, A., Sethi, R. and Ullman, J.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley (1986).

3) Alpern, B., Wegman, M. and Zadeck, F.: Detecting equality of variables in programs, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*[1)], pp.1–11 (1988).
4) Appel, A.: *Compiling with Continuations*, Cambridge University Press (1992).
5) Appel, A.: *Modern Compiler Implementation in ML*, preliminary edition, Cambridge University Press (1997).
6) Arsac, J. and Kodratoff, Y.: Some techniques for recursion removal from recursive functions, *ACM Trans. Prog. Lang. Syst.*, Vol.4, No.2, pp.295–322 (1982).
7) Arvind, Nikhil, R. and Pingali, K.: I-structures: Data structures for parallel computing, *ACM Trans. Prog. Lang. Syst.*, Vol.11, No.4, pp.598–632 (1989).
8) Auslander, M. and Strong, H.: Systematic recursion removal, *Comm. ACM*, Vol.21, No.2, pp.127–134 (1978).
9) Ayers, J.: Recursive programming in Fortran II, *Comm. ACM*, Vol.6, No.11, pp.667–668, (1963).
10) Bauer, F. and Wössner, H.: *Algorithmic Language and Program Development*, Texts and Monographs in Computer Science, Springer-Verlag (1982).
11) Bird, R.: Improving programs by the introduction of recursion, *Comm. ACM*, Vol.20, No.11, pp.856–863 (1977).
12) Boiten, E.: The many disguises of accumulation, Technical Report 91-26, Department of Informatics, University of Nijmegen (1991).
13) Boiten, E.: Improving recursive functions by inverting the order of evaluation, *Science of Computer Programming*, Vol.18, No.2, pp.139–179 (1992).
14) Burstall, R. and Darlington, J.: A transformation system for developing recursive programs, *J. ACM*, Vol.24, No.1, pp.44–67 (1977).
15) Cheng, P., Harper, R. and Lee, P.: Generational stack collection and profile-driven pretenuring, *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'98*), Davidson, J. (Ed.), pp.162–173, Montreal, Canada (1998).
16) Colussi, L.: Recursion as an effective step in program development, *ACM Trans. Prog. Lang. Syst.*, Vol.6, No.1, pp.55–67 (1984).
17) Cooper, D.: The equivalence of certain computations, *Comput. J.*, Vol.9, No.1, pp.45–52 (1966).
18) Darlington, J. and Burstall, R.: A system which automatically improves programs, *Acta Inf.*, Vol.6, No.1, pp.41–60 (1976).
19) Friedman, D., Wand, M. and Haynes, C.: *Essentials of Programming Languages*, McGraw Hill (1992).
20) Futamura, Y., Konishi, Z. and Glück, R.: Implementation of an experimental system for automatic program transformation based on generalized partial computation, *Proc. Third International Workshop on Intelligent Software Engineering* (*WISE*^{3}), Menzies, T. (Ed.), pp.39–48, Limerick, Ireland (2000).
21) Futamura, Y. and Nogi, K.: Generalized partial computation, *Partial Evaluation and Mixed Computation*, Ershov, A., Bjørner, D. and Jones, N. (Ed.), North-Holland (1988).
22) Futamura, Y., Nogi, K. and Takano, A.: Essence of generalized partial computation, *Theor. Comput. Sci.*, Vol.90, pp.61–79 (1991).
23) Futamura, Y. and Otani, H.: Recursion removal rules for linear recursive programs and their effectiveness (in Japanese), *Computer Software*, Vol.15, No.3, pp.38–49 (1998).
24) Harrison, P. and Khoshnevisan, H.: A new approach to recursion removal, *Theor. Comput. Sci.*, Vol.93, No.1, pp.91–113 (1992).
25) Hughes, R.: A novel representation of lists and its application to the function "reverse", *Inf. Process. Lett.*, Vol.22, No.3, pp.141–144 (1986).
26) Jones, N.: An introduction to partial evaluation, *ACM Comput. Surv.*, Vol.28, No.3, pp.480–504 (1996).
27) Jones, N., Gomard, C. and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*, Prentice-Hall (1993).
28) Knuth, D.: *Fundamental Algorithms*, third edition, *The Art of Computer Programming*, Vol.1, Addison-Wesley (1997).
29) Konishi, Z.: Personal communication (Oct. 2000).
30) Liu, Y. and Stoller, S.: From recursion to iteration: What are the optimizations? *2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (*PEPM'00*), Lawall, J. (Ed.), pp.73–82, Boston, Massachusetts, Jan. 22–23 (2000).
31) Liu, Y., Stoller, S. and Teitelbaum, T.: Static caching for incremental computation, *ACM Trans. Prog. Lang. Syst.*, Vol.20, No.3, pp.546–585 (1998).
32) Minamide, Y.: A functional representation of data structures with a hole, *Proc. 25th ACM Symposium on Principles of Programming Languages*, pp.75–84, San Diego, CA (1998).
33) Paige, R. and Koenig, S.: Finite differencing of computable expressions, *ACM Trans. Prog. Lang. Syst.*, Vol.4, No.3, pp.402–454 (1982).
34) Partsch, H.: *Specification and Transformation of Programs*, Texts and Monographs in Computer Science, Springer-Verlag (1990).

35) Paterson, M. and Hewitt, C.: Comparative schematology, Artificial Intelligence Memo, No.201, AI Lab, MIT (1970).
36) Pettorossi, A.: Program development using lambda abstraction, *Seventh Conference of Foundations of Software Technology and Theoretical Computer Science, Proceedings*, Nori, K. (Ed.), *Lecture Notes in Computer Science*, Vol.287, pp.420–434, Springer-Verlag (1987).
37) Pettorossi, A. and Proietti, M.: Rules and strategies for transforming functional and logic programs, *ACM Comput. Surv.*, vol.28, No.2, pp.360–414 (1996).
38) Pettorossi, A. and Skowron, A.: Higher order generalization in program derivation, *TAPSOFT '87: Proc. International Joint Conference on Theory and Practice of Software Development,* Vol.2: *Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Functional and Logic Programming and Specifications* (*CFLP*), Ehrig, H., Kowalski, R., Levi, G. and Montanari, U. (Eds.), *Lecture Notes in Computer Science*, Vol.250, pp.182–196, Springer-Verlag (1987).
39) Rosen, B., Wegman, M. and Zadeck, F.: Global value numbers and redundant computations, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*[1], pp.12–27 (1988).
40) Schorr, H. and Waite, W.: An efficient machine-independent procedure for garbage collection in various list structures, *Comm. ACM*, Vol.10, No.8, pp.501–506 (1967).
41) Sørensen, M., Glück, R. and Jones, N.: A positive supercompiler, *J.Functional Programming*, Vol.6, No.6, pp.811–838 (1996).
42) Stering, L. and Shapiro, E.: *The Art of Prolog: Advanced Programming Techniques*, second edition, MIT Press (1994).
43) Strong, Jr., H.: Translating recursion equations into flow charts, *J. Comput. Syst. Sci.*, Vol.5, No.3, pp.254–285 (1971).
44) Turchin, V.: The concept of a supercompiler, *ACM Trans. Prog. Lang. Syst.*, Vol.8, No.3, pp.292–325 (1986).
45) Wadler, P.: Deforestation: Transforming programs to eliminate trees, *Theor. Comput. Sci.*, Vol.73, No.2, pp.231–248 (1990).
46) Walker, S. and Strong, H.: Characterizations of flowchartable recursions, *J. Comput. Syst. Sci.*, Vol.7, No.4, pp.404–447 (1973).
47) Wand, M.: Continuation-based program transformation strategies, *J. ACM*, Vol.27, No.1, pp.164–180 (1980).
48) Waters, R.: Automatic transformation of series expressions into loops, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.1, pp.52–98 (1991).
49) Wilhelm, R. and Maurer, D.: *Compiler Design*, International Computer Science Series, Addison-Wesley (1995).

(Received October 25, 2000)
(Accepted February 20, 2001)



Kazuhiko Kakehi is a Ph.D. student of Information and Computer Science at Waseda University. He received his B.S. and M.S. degrees from Waseda University in 1997 and 1999, respectively. Currently he receives the Research Fellowship of the Japan Society for the Promotion of Science (JSPS) for Young Scientists. His main research interests are partial evaluation and program transformation in functional frameworks, as well as memory management like garbage collection.

Robert Glück is an associate professor of Computer Science at the University of Copenhagen. He received his M.Sc. and Ph.D. degrees in 1986 and 1991 from the University of Technology in Vienna, where he also worked as assistant professor. He received his Habiliation (venia docendi) in 1997. He was research assistant at the City University of New York and received twice the Erwin-Schrödinger-Fellowship of the Austrian Science Foundation. After being an Invited Fellow of JSPS at Waseda University in Tokyo, he is now funded by Japan Science and Technology Corporation (JST). His main research interests are advanced programming languages, theory and practice of program transformation, and metaprogramming techniques.

Yoshihiko Futamura is Professor of Department of Information and Computer Science, and the director of the Institute for Software Production Technology (ISPT) of Waseda University. He received his B.S. degree in mathematics from Hokkaido University in 1965, M.S. degree in applied mathematics from Harvard University in 1972 and Ph.D. degree from Hokkaido University in 1985. He joined Hitachi Central Research Laboratory in 1965 and moved to Waseda University in 1991. He was a visiting professor of Uppsala University from 1985 to 1986 and a visiting scholar of Harvard University from 1988 to 1989. Automatic generation of computer programs and programming methodology are his main research fields. He is the inventor of the Futamura Projections in partial evaluation and ISO8631 PAD (Problem Analysis Diagram).