New Generation Computing, 9 (1991) 135-147
OHMSHA, LTD. and Springer-Verlag

Preliminary Arrangements of Arguments in Lazy Evaluation*

Liao HUSHENG
Department of Software Engineering,
Beijing Computer Institute,
No. 24 Xi-San-Huan-Bei Road, Beijing, China.
Masato TAKEICHI
Department of Mathematical Engineering,
Faculty of Engineering,
University of Tokyo,
Bunkyo-ku, Tokyo 113, Japan.

Received 8 June 1987
Revised manuscript received 3 September 1990

Abstract This paper describes a new implementation technique called *preliminary arrangements* of arguments for lazy functional languages. Unlike conventional lazy evaluators, the evaluator with preliminary arrangements partly processes every argument before calling functions. It works in a lazy way with less memory cells than conventional methods. The practical importance of this technique is demonstrated by some benchmark results.

Keywords: Functional Language, Lazy Evaluation, SECD Machine

§1 Introduction

Functional languages are known primarily for their elegance, clearity and expressive power. Lazy evaluation has an advantage in that it performs only the necessary computational steps for executing programs. Implementation methods of lazy evaluators have been studied by many researchers. Henderson[2,3] describes such an implementation based on the classical *SECD machine*. Takeichi[11] proposes another SECD-based machine *FLFM* of which code can be considered as an intermediate code for generating machine instructions of

* Research partially done while both authors were at the University of Electro-Communications, Tokyo. The work of the second author is partially supported by Grant-in-Aid for Scientific Research #01550278 of The Ministry of Education, Science and Culture, Japan.

conventional machines. Experiments show, however, that execution of functional programs on such an SECD machine is expensive both in running time and memory space. When an applicative expression of the form (f a_1 ... a_n) is evaluated in a lazy way, evaluation of the arguments a_1, ..., a_n should be delayed until their values are actually needed in the function f. To achieve this with the SECD machine, the code and the environment for each argument are kept in a structure called *closure*. A straightforward application of this strategy produces a lot of closure cells as the computation proceeds. What is worse is that they should remain existent for a long period because the time when the argument is to be evaluated is not known in advance. This is one of the reasons why many implementations of lazy functional languages adopt another strategy called *combinator graph reduction*.

An implementation technique based on combinatory logic and graph reduction was first presented by Turner[13] and exploited in various ways by many researchers as summarised in Ref. 9). The basic idea is to represent an expression by a combinator graph and to reduce them to get a simplified expression as the result. It is well-known fact that any expression can be translated into a combinator expression consisting solely of the combinators S and K. Combinators are considered as specifically chosen functions of the closed form, i.e., not containing any free variables. Hence the translation process effectively eliminates program variables and the evaluation process does not require the environment that associates variables with values. Although Turner used only a fixed set of combinators S, K, I, B, C, and with additional operations, Hughes[4] extends this to introduce new combinators, called *super-combinators*, depending on programs. In either case, combinators have their own reduction rules. The evaluator is, therefore, realised by making the combinator graph be rewritten according to the rules. Graph reduction will be most easily performed by an interpreter which traverses and transforms the graph repeatedly. If we want to produce fixed code for conventional computers, reduction process can be compiled into machine code as described in Ref. 6) or in Ref. 10). A similar implementation method was taken in the Lazy ML compiler.[1,5] The Lazy ML compiler does not exploit the property of *full-laziness* which is a very particular feature of the combinator reduction scheme, while full-laziness can be realised in the SECD scheme as well.[12]

Both of the above implementation techniques seem very different in nature, but they have some similarities. One of them is the use of dynamic structures for realising the *call-by-need* mechanism by which recomputation is avoided for the same expression. The SECD approach offers the environment cell for that purpose; it might be shared by different occurrences of the same variable and it is considered as a node of a graph for variable references. In the graph reduction scheme, and part of the combinator graph may possibly be shared as part of larger expression graphs. How efficiently such structures are dealt with is, therefore, important in both implementation methods.

It is still arguable which of these methods is more advantageous over the other for conventional computers. Peyton Jones[8] states that the combinator reduction wins in time and in space over lambda reducers using the environment model, whereas experiments show that the reverse holds if measured by executing fixed code on several machines.[11] In this paper we propose a technique called *preliminary arrangements* of arguments for saving the memory space for the closure and the environment link in lazy evaluators based on the SECD scheme. The evaluator arranges arguments in a structure that consumes less memory space than closures. It preserves laziness implemented by ordinary lazy evaluators. What is improved by preliminary arrangements of arguments is the reduction of memory space. It is most attractive in practice that large programs can be executed efficiently as we shall see in Section 5.

§2 Preliminary Arrangements

We introduce here the basic idea of preliminary arrangements of arguments. Consider a combination, i.e., a function call

$$f\ a$$

where f is a function defined elsewhere as

$$f\ x = \ldots\ x\ \ldots$$

and a is an expression possibly containing variables of which values are kept in an environment list E.

According to the ordinary lazy evaluation scheme, the combination $(f\ a)$ is translated into following code:

- Make a closure$[C_a: E]$ for a where C_a is the code for evaluating a.
- Push it onto the stack.
- Call f.

The function body works as follows:

- Pop up the argument bound to x from the stack.
- Extend the environment list E to create a new environment E_f for f.
- Evaluate the body of f.

When computation proceeds to the argument associated with x in the function body, the closure$[C_a: E]$ is picked up from E_f to evaluate the expression a in the environment E.

What we are interested here is the lifetime of the environment E. Assume that the argument a is simply a variable a. It is easy to see that the closure$[C_a: E]$ is not necessary. We may push the *value* associated with a in E onto the stack. Thus the environment E becomes free in this case even if E conveys values for other variables. This technique is rather standard in implementing lazy evaluators based on the SECD scheme.[3,11]

Syntactic domains

$b \in \mathbf{Bas}$	basic values
$x \in \mathbf{Ide}$	identifiers
$e \in \mathbf{Exp}$	expressions

Abstract syntax

$e ::= b \mid x \mid e\ e \mid \mathbf{fn}\ x.\ e$

Semantic domains

$\mathbf{B}$	basic values
$\mathbf{E} = [\mathbf{B} \to \mathbf{F} + \mathbf{P}]_+$	expressible values
$\mathbf{F} = \mathbf{D} \to \mathbf{E}$	functions
$\mathbf{P} = \mathbf{E} \times \mathbf{E}$	pairs (constructed by primitive constructors)
$\mathbf{D} = \mathbf{E}$	denotable values
$\mathbf{U} = \mathbf{Ide} \to \mathbf{D}_\perp$	environments

Semantic functions

$B: \mathbf{Bas} \to \mathbf{B}$ (unspecified)

$E: \mathbf{Exp} \to \mathbf{U} \to \mathbf{E}$

$$E[\![b]\!]\rho = B[\![b]\!]$$
$$E[\![x]\!]\rho = \rho[\![x]\!]$$
$$E[\![e_0\ e_1]\!]\rho = (E[\![e_0]\!]\rho)(E[\![e_1]\!]\rho)$$
$$E[\![\mathbf{fn}\ x.\ e_0]\!]\rho = \lambda\delta.\ E[\![e_0]\!](\rho + \langle x \to \delta \rangle)$$

Notations

Domain construcion operator + stands for the disjoint sum.

For any domain $\mathbf{X}$, $\mathbf{X}_\perp = \mathbf{X} + \{\mathbf{err}\}$.

For an environment ρ, $\rho + \langle x \to \delta \rangle$ denotes

$$\lambda y.\ \mathbf{if}\ x = y\ \mathbf{then}\ \delta\ \mathbf{else}\ \rho[\![y]\!].$$

Initial Environment

The initial environment ρ_0 satisfies $\rho_0[\![x]\!] \neq \mathbf{err}$ for pre-defined identifiers x.

Representation of values

$\mathbf{V} = \mathbf{B} + \mathbf{P} + \mathbf{C} + \mathbf{H}$	value representation
$\mathbf{B}$	basic values
$c \in \mathbf{P}$	pairs (list cells)
$\gamma \in \mathbf{C} = \mathbf{Exp} \times \mathbf{Env}$	closures
$\theta \in \mathbf{H} = \mathbf{V} \times \mathbf{V}$	half-cooked combinations
$\pi \in \mathbf{Env} = \mathbf{Ide} \to \mathbf{V}$	(dynamic) environments

Preliminary arrangement rules

$P: [\mathbf{Exp} + \mathbf{C} + \mathbf{H}] \to \mathbf{Env} \to \mathbf{V}$

$P[\![b]\!]\pi = B[\![b]\!]$	basic value
$P[\![c]\!]\pi = c$	list cell
$P[\![x]\!]\pi = \pi[\![x]\!]$	value in the environment
$P[\![e_0\ e_1]\!]\pi = \langle P[\![e_0]\!]\pi;\ P[\![e_1]\!]\pi \rangle$	half-cooked combination
$P[\![\mathbf{fn}\ x.\ e_0]\!]\pi = [\mathbf{fn}\ x.\ e_0 :\ \pi]$	closure
$P[\![\gamma]\!]\pi = \gamma$	
$P[\![\theta]\!]\pi = \theta$	

Fig. 1 Specification of a simple language and the preliminary arrangement rules.

Generalisation of this idea leads to our preliminary arrangement technique. We introduce a new structure called *half-cooked combination* for constituents of the compound expression $(e_0\ e_1 \ldots e_n)$. The preliminary arrangement rule P for a small language is defined in Fig. 1. Although the language is very simple, fundamental features of functional languages are included; functional applications by combination, and abstractions by **fn**-expressions*. Although the rules with local declarations such as

* We avoid to use λ in the source language because it is used in the description of the semantics. We use **fn** instead.

e_0 **where** $x_1 = e_1$ **and**...**and** $x_n = e_n$

and

e_0 **whererec** $x_1 = e_1$ **and**...**and** $x_n = e_n$

are straightforward, we omit here for brevity.

We denote the half-cooked combination as

$\langle \varepsilon_0; \varepsilon_1 \rangle$.

We also write

$\langle \varepsilon_0\ \varepsilon_1 ... \varepsilon_n \rangle$

as an abbreviation of

$\langle \langle \varepsilon_0\ \varepsilon_1 ... \varepsilon_{n-1} \rangle; \varepsilon_n \rangle$.

Although the function P in Fig. 1 denotes the meaning of preliminary arrangement, operational description should be presented for explaining how it works in evaluation.

Assume that we have a combination of the form $(e_0\ e_1)$, in which e_0 and e_1 are expressions. Consider first the argument e_1.

- If e_1 is a basic value or a data structure, it is passed to the function as it is.
- If e_1 is a variable x, the value $\pi[x]$ associated with x is fetched from the current environment to yield ε_1. Note that this *does not* mean the evaluation of x even if a *suspended expression* is associated to x. The suspended expression is represented by a half-cooked combination.
- If e_1 is a combination, apply the rules recursively to yield a half-cooked combination.
- If e_1 is an **fn**-abstraction, make a closure cell which consists of the function itself with the current environment.
- If e_1 is already a closure or a half-cooked combinations, it remains as it is. Note that the latter case does not occur unless we take account of the selector operations of the pair. They are, of course, included in the actual evaluator.

The same rules may be applied to e_0 except that the first case should be treated as illegal. These rules may be applied to each argument e_i of a more general form of expression $(e_0\ e_1\ ...\ e_n)$ yielding $\langle \varepsilon_0\ \varepsilon_1\ ...\ \varepsilon_n \rangle$.

What is needed when evaluation proceeds to force the half-cooked combination is rather simple. Evaluation of a half-cooked combination $\langle \varepsilon_0\ \varepsilon_1 ... \varepsilon_n \rangle$ proceeds as follows:

- Push ε_n, ..., ε_1 onto the stack.
- Evaluate ε_0 and call the function.

§3 Storage Allocation

The ordinary lazy SECD machine allocates memory cells in the heap storage to hold various objects such as functions, variables, and code for execution. Some implementation places the code and the stack in separate area of storage. In either case, three kinds of data structures are allocated in the heap: *list cells* produced by programs, *environment link cells* which associate variables with arguments, and *closure cells* each consisting of a code address and a pointer to the environment cell.

We are required to implement the half-cooked combination with a new type of data structure. A simple method is to represent

$$\langle \varepsilon_0\ \varepsilon_1\ \ldots\ \varepsilon_n \rangle$$

by a linear list using $(n + 1)$ binary cells in an obvious way.

We now consider a realistic example for estimating the amount of long-lived cells. Given a value s and two lists x and y, $(lookup\ s\ x\ y)$ tests the elements of x and y in sequence until s is found in x, and returns the corresponding element of y:

$$lookup\ s\ x\ y =$$
$$\quad \textbf{if}\ s = car\ x\ \textbf{then}\ car\ y\ \textbf{else}\ lookup\ s\ (cdr\ x)(cdr\ y).$$

We assume here that the functions *car* and *cdr* are primitive functions which select components of list structures. Although such primitives would be coded in-line by the compiler, we take them as ordiary functions for explanatory purposes. Consider an expression $(lookup\ 6\ [2;4;6;8](from\ 11))$ where

$$from\ n = n : from\ (n + 1)$$

Note that the second list, i.e., $(from\ 11)$ denotes an infinite list and must be evaluated lazily for the expression to be meaningful. We shall write down the computational process using a similar notation as the source language; parameter bindings are represented by **with**-clauses. Each binding is implemented with an environment link cell.

We first trace the steps performed by an evaluator which passes arguments to functions in an ordinary manner.

$$\begin{aligned}
&lookup\ 6\ [2;4;6;8](from\ 11)\\
\rightarrow\ &\textbf{if}\ s = car\ x\ \textbf{then}\ car\ y\ \textbf{else}\ lookup\ s\ (cdr\ x)(cdr\ y)\\
&\textbf{with}\ s = 6\ \textbf{and}\ x = [2;4;6;8]\ \textbf{and}\ y = from\ 11 \quad \{E_1\}\\
\rightarrow\ &lookup\ s\ (cdr\ x)(cdr\ y)\\
&\textbf{with}\ s = 6\ \textbf{and}\ x = [2;4;6;8]\ \textbf{and}\ y = from\ 11 \quad \{E_1\}\\
\rightarrow\ &\textbf{if}\ s = car\ x\ \textbf{then}\ car\ y\ \textbf{else}\ lookup\ s\ (cdr\ x)(cdr\ y)\\
&\textbf{with}\ s = 6\ \textbf{and}\ x = cdr\ x\ \textbf{and}\ y = cdr\ y \quad \{E_2\}\\
&\quad \textbf{with}\ s = 6\ \textbf{and}\ x = [2;4;6;8]\ \textbf{and}\ y = from\ 11 \quad \{E_1\}
\end{aligned}$$

→ *lookup s* (*cdr x*)(*cdr y*)
with *s* = 6 **and** *x* = [4;6;8] **and** *y* = *cdr y* {E_2}
with *s* = 6 **and** *x* = [2;4;6;8] **and** *y* = *from* 11 {E_1}
→ **if** *s* = *car x* **then** *car y* **else** *lookup s* (*cdr x*)(*cdr y*)
with *s* = 6 **and** *x* = *cdr x* **and** *y* = *cdr y* {E_3}
with *s* = 6 **and** *x* = [4;6;8] **and** *y* = *cdr y* {E_2}
with *s* = 6 **and** *x* = [2;4;6;8] **and** *y* = *from* 11 {E_1}
→ *car y*
with *s* = 6 **and** *x* = [6;8] **and** *y* = *cdr y* {E_3}
with *s* = 6 **and** *x* = [4;6;8] **and** *y* = *cdr y* {E_2}
with *s* = 6 **and** *x* = [2;4;6;8] **and** *y* = *from* 11 {E_1}
...
→ 13

Notice that the variable x in the environment becomes to have a list rather than a closure once it has been evaluated. This is a consequence of the call-by-need mechanism. The last reduction steps for obtaining 13 from the expression (*from* 11) are omitted here since these are irrelevant to our discussion.

The environment has a structure of which component cells hold values of local variables. Evaluation of (*car y*) is performed using the environment in Fig. 2. The environment link cell is shown by a concrete cell, and the closure by a dotted cell. It may be observed that only the values for three instances of the variable y need be kept for computing (*car y*), and the value for instances of the other variables may be thrown away. Allocation of environment cells for all the bindings causes extra consumption of memory cells.

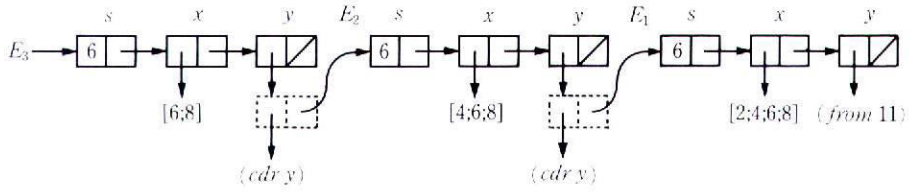

Fig. 2 Cells created by an ordinary lazy evaluator.

On the other hand, the computation with preliminary arrangements works as follows:

lookup 6[2;4;6;8](*from* 11)
→ **if** *s* = *car x* **then** *car y* **else** *lookup s* (*cdr x*)(*cdr y*)
with *s* = 6 **and** *x* = [2;4;6;8] **and** *y* = ⟨*from* 11⟩ {E_1}
→ *lookup s* (*cdr x*)(*cdr y*)
with *s* = 6 **and** *x* = [2;4;6;8] **and** *y* = ⟨*from* 11⟩ {E_1}
→ **if** *s* = *car x* **then** *car y* **else** *lookup s* (*cdr x*)(*cdr y*)
with *s* = 6 **and** *x* = [4;6;8] **and** *y* = ⟨*cdr*⟨*from* 11⟩⟩ {E_2}
→ *lookup s* (*cdr x*)(*cdr y*)

with $s = 6$ **and** $x = [4;6;8]$ **and** $y = \langle cdr \langle from\ 11 \rangle \rangle$ $\{E_2\}$
→ **if** $s =$ car x **then** *car* y **else** *lookup* s $(cdr\ x)(cdr\ y)$
with $s = 6$ **and** $x = [6;8]$ **and** $y = \langle cdr \langle cdr \langle from\ 11 \rangle \rangle \rangle$ $\{E_3\}$
→ *car* y
with $s = 6$ **and** $x = [6;8]$ **and** $y = \langle cdr \langle cdr \langle from\ 11 \rangle \rangle \rangle$ $\{E_3\}$
...
→ 13

The environment for (*car* y) in this case is shown in Fig. 3. The half-cooked combination cell is depicted as a dashed cell.

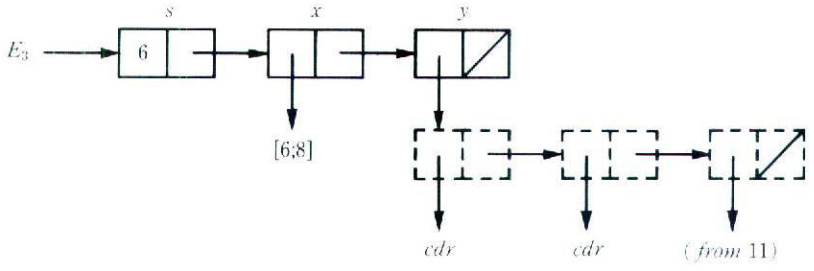

Fig. 3 Cells created by preliminary arrangements.

It can be seen that half-cooked combinations have taken the place of closures for delayed computation. The number of cells required for evaluating (*lookup* 6 [2;4;6;8] (*from* 11)) using our preliminary arrangement technique is $(n + 3)$ compared with $(4n - 1)$ by the conventional method. An important role of the half-cooked combination is to cut off the connection between closures and environments so that unnecessary environment cells become garbage. By preliminary arrangements, the environment for a combination is freed at the beginning of evaluation of the function body, while in conventional methods it should be saved until all of the arguments are evaluated.

There is a slight drawback, however. Since preliminary arrangements are carried out before evaluation of the function body, all of the arguments are arranged as described in Section 2 whether they need be evaluated or not. Notice that the arguments are *not yet evaluated*, but part of housekeeping tasks in evaluation is performed in the arrangement stage. In addition to the overhead for creating the half-cooked cell, preliminary arrangements for every argument may reduce the speed of execution. Although the total efficiency depends on storage management techniques, we reasonably expect that fewer chances of garbage collection make up for the loss. We shall demonstrate the practical importance of preliminary arrangements in Section 5.

§4 More on Preliminary Arrangements

As pointed out by many researchers, lazy evaluation sometimes suffers from the shortage of memory cells. A serious problem may arise in conventional methods when the program yields a large data structure. Every element of the data structure creates a closure that will be reduced to some simple value. Such a closure cell contains a pointer to its own environment that should be existent until the closure is reduced to a non-closure value. Another source of long-lived cells is the use of local declarations and higher order functions, both of which are important features in functional programming. Lazy evaluation suspends evaluation of any expression as long as possible, at least in principle. Expressions with local declarations necessarily generate environments that remain existent until all of the local expressions are completely evaluated. The situation of higher order functions is very similar to this case.

It becomes clear from the example in the previous section that some of the long-lived cells can be detected at creation time with little effort. Some of the arguments can be evaluated before the call without changing the meaning of the program. Such an optimisation technique becomes a standard method for lazy functional languages. It is called *strictness analysis* which was first presented by Mycroft[7] and has been used extensively in many implementations. In general, however, it is difficult to know in advance whether the environment will be used or not for non-strict arguments. Our approach to the problem of suspended arguments is to arrange them preliminarily before calling functions. The preliminary arrangement technique requires only slight modification of the evaluator, but no other time-consuming analysis of the program. We may apply preliminary arrangements to the *cons* function for constructing a list. It returns a list consisting of several half-cooked combinations as a value, instead of one with closures. Since the number of cells required for combinations is much less than that for closures in conventional method, this makes an improvement on storage usage.

We can make some optimisations for arguments during preliminary arrangements. Consider the case that the variable n is assigned a number in the current environment. In such a case, we can immediately get its result without creating half-cooked combinations. For arguments of strict functions such as arithmetic operations, delayed evaluation is not required because all the arguments need be evaluated sooner or later. We can deal with $(car\ e)$ in a similar way if the variable e is assigned a list cell.

$$P[car\ e]\pi = \begin{cases} a & \text{if } P[e]\pi = (a, b) \\ \langle car;\ P[e]\pi \rangle & \text{otherwise} \end{cases}$$

where (a, b) is a pair constructed by *cons* from a and b. We can apply similar optimisation techniques to arguments of the form $(cdr\ e)$, $(null\ e)$ or $(atom\ e)$. In either case, e is not evaluated until it is forced; only the form of e is tested

in this stage. It should be noted that the function is still evaluated lazily, while the arguments are processed as far as possible before calling the function. This is considered as an extension of the preliminary arrangement rule for variables, where the associated value is fetched but it is not evaluated.

§5 Experimental Results

In order to estimate the applicability of the preliminary arrangement technique, we have made a compiler with preliminary arrangements for the *Fully Lazy Functional Machine*, a variant of the SECD machine.[11] It is based on a compiler for the ordinary lazy evaluator generating FLFM code. The FLFM code generated by two compilers is then converted to conventional machine instructions. The new compiler translates every argument and local declaration according to the preliminary arrangement rules. Also included are the optimisation rule for arithmetic functions and for *car*, *cdr*, *null*, and *atom* as described in the previous section. Total running time including time for garbage collection was measured on the Sun-2 workstation with different heap sizes. Four programs are written in *uc*,[11] a functional language of which syntax is borrowed in part from Miranda.[14] See the listing in Appendix. The differences between object programs with and without preliminary arrangements are shown in Table 1.

Table 1 Run-time in seconds for four programs.

Program	Without preliminary arrangements (sec.)				With preliminary arrangements (sec.)			
	Heap size (cells)				Heap size (cells)			
	3k	10k	30k	100k	3k	10k	30k	100k
nFib 20	5.0	4.9	4.9	4.8	3.4	3.4	3.4	3.4
Ram 10	—	21.8	17.0	16.8	21.5	13.9	12.9	12.9
Ack 3 5	16.1	11.3	10.5	10.5	10.2	8.0	7.6	7.6
Ackf 3 5	—	—	9.6	4.2	8.0	4.8	4.1	4.1

The first example program (*nFib* 20) consumes no more than 3000 cells and forces no garbage collection in either case. The optimisation based on preliminary arrangements of arithmetic functions brings 30% improvement on running time. The second program (*Ram* 10) produces many list cells besides house-keeping cells, i.e., environment link, closure, and half-cooked combination cells. Without preliminary arrangements, the program cannot be executed on 3000 heap cells. It is observed from the second and the third programs that the improvement ratio is higher than 30% when garbage collection is taken place. Although it depends on the efficiency of the garbage collector, this implies that the time required for garbage collection must not be overlooked. It is concluded that the preliminary arrangement technique make some 30% improvement without garbage collection time, and more if garbage collection time is taken into account. For the fourth program (*Ackf* 3 5) dealing with higher order functions, the effect of preliminary arrangements is remarkable. A great

many closure cells are produced by the conventional method. The time for calling functions is comparable when the heap is large enough to execute the program without garbage collection.

The benchmark results lead us to believe that preliminary arrangements gain more than the overhead incurred by additional tasks before calling functions.

§6 Conclusions

We believe that the preliminary arrangement technique is effective for implementation of lazy functional languages. In particular, programs which use higher order functions and complex data structures run considerably faster by preliminary arrangements. The use of preliminary arrangements alleviates the conflict between the limited storage resource and the need of memory space for suspended arguments.

Our compiler with preliminary arrangements was implemented on the FLFM machine. The structure of the FLFM machine is not fully suited for preliminary arrangements. We should develop a new functional machine accommodated to the preliminary arrangement technique. From the viewpoint of optimisation, argument arrangement for strict functions has become a standard technique and is promising.[9] It may be possible to extend the idea and deal with user-defined functions by *strictness analysis* so that finite data structures are evaluated in a call-by-value fashion. Our preliminary arrangement technique deals with *non-strict* arguments which must be evaluated lazily. It is, therefore, considered as a new parameter mechanism which retains non-strict semantics. We hope that this research has made a step toward an efficient implementation of functional languages.

Acknowledgements

We sincerely thank NGC referees for many constructive criticism and suggestions on earlier draft of the paper. Their advices have been invaluable in improving the technical content.

References

1) Augustsson, L., "A compiler for Lazy ML," *Proc. 1984 ACM Symp. Lisp and Functional Programming*, pp. 218-227, 1984.
2) Henderson, P., *Functional Programming: Application and Implementation*, Prentice-Hall, 1980.
3) Henderson, P., Jones, G. A. and Jones, S. B., *The Lispkit Manual, Technical Monograph PRG-32*, Oxford University Computing Laboratory, 1983.
4) Hughes, R. J. M., "Super-combinators: a New Implementation Method for Applicative Languages," *Proc. 1982 ACM Symp. Lisp and Functional Programming*, pp. 1-10, 1982.

5) Johnsson, T., "Efficient Compilation of Lazy Evaluation," *Proc. SIGPLAN'84 Symp. on Compiler Construction*, pp. 58-69, 1984.
6) Jones, N. D. and Muchnick, S. S., "A Fixed-Program Machine for Combinator Expression Evaluation," *Proc. 1982 ACM Symp. LISP and Functional Programming*, pp. 11-20, 1982.
7) Mycroft, A., "Abstract Interpretation and Optimising Transformations for Applicative Programs," *Ph. D. Thesis*, University of Edinburgh, 1981.
8) Peyton Jones, S. L., "An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions," *Proc. 1982 ACM Symp. LISP and Functional Programming*, pp. 150-158, 1982.
9) Peyton Jones, S. L.: *The Implementation of Functional Programs*, Prentice-Hall, 1987.
10) Takeichi, M., "An Alternative Scheme for Evaluating Combinator Expressions," *Journal of Information Processing*, 7, pp. 246-253, 1985.
11) Takeichi, M., "Fully Lazy Evaluation of Functional Programs," *D. Eng. Thesis*, University of Tokyo, 1987.
12) Takeichi, M., "Lambda-Hoisting: a Transformation Technique for Fully Lazy Evaluation of Functional Programs," *New Generation Computing*, 5, pp. 377-391, 1988.
13) Turner, D. A., "A New Implementation Technique for Applicative Languages," *Software-Practice and Experience*, 9, pp. 39-49, 1979.
14) Turner, D. A., "Miranda: A Non-strict Functional Language with Polymorphic Types," *Functional Programming Languages and Computer Architecture*, *Lecture Notes in Computer Science*, 201, Springer-Verlag, pp. 1-16, 1985.

Appendix

Benchmark programs written in *uc*.

```
# Counting function calls.
nFib 20
   whererec nFib n = if n < 2 then 1
                               else nFib (n − 1)+ nFib (n − 2)+ 1

# Finding 10 Ramanujan's numbers that are equal to two different
# sums of two natural numbers raised to the third power.
Ram 10
   whererec Ram n = take n (ram (sort 1))
   and ram (x:(y: z))= if cubes x = cubes y then (x, y): ram (y: z)
                                            else ram (y: z)
   and sort k =(k, k): merge[(k, b)| b <−[k + 1..]](sort (k + 1))
   and merge (x: u)(y: v)=
     if cubes x ≥ cubes y then y: merge (x: u) v else x: merge u (y: v)
   and cubes (a, b)= a*a*a + b*b*b

# Ackermann's function
Ack 3 5
   whererec Ack m n = if m = 0 then n + 1 else
            if n = 0 then Ack (m − 1) 1
                     else Ack (m − 1)(Ack m (n − 1))

# Ackermann's function using functional representation of integers
Ackf 3 5
```

```
whererec Ackf m n = suback (rep m)(rep m)(+ 1) 0
and suback m = m aug succ
and aug h n = n h (h one)
and rep n = if n = 0 then zero else succ (rep (n − 1))
and succ n f x = f (n f x)
and one f x = f x
and zero f x = x
```