A Functional Machine for Fully Lazy Evaluation
(Extended Abstract)

Masato Takeichi

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

ABSTRACT

In order to explore the applicability of full laziness in functional programming, we have developed an abstract machine called Fully Lazy Functional Machine (FLFM). The machine is a variant of the SECD machine suitable for evaluating expressions written in Fully Lazy Lisp (FLL). This paper describes the structure of FLFM and algorithms for translating functional languages into an intermediate language FLL, and for compiling FLL expressions into FLFM code which can be executed using a small interpreter. It is possible, however, to generate code for conventional computers from FLFM programs to gain efficiency. The implementation technique described in this paper is useful to generate efficient code for a wide class of functional languages. Actual implementation on the MC68000 is also described.

1. Introduction

Hughes [7] introduces so-called fully lazy evaluation of applicative expressions in relation with combinators. Full laziness implies ordinary laziness in [3,4]. Among others, it has an important property that every expression is evaluated at most once, whereas in ordinary lazy evaluation scheme only the expression passed as argument to function is evaluated at most once. Hughes also describes an algorithm that translates applicative expressions with lambda abstraction into fully lazy form. Takeichi [11] describes an extended algorithm that deals with expressions with local declarations. We will call this process *lambda-hoisting* after the code-hoisting technique [1]. A similar but different transformation technique called lambda-lifting is described in [9].

We begin with a review of the translation algorithm to clarify our intention to design a functional machine for fully lazy evaluation. In Section 2 we will specify an algorithm that translates *Lispkit Lisp* [5,6] expressions into *Fully Lazy Lisp* (FLL) expressions. A machine model for fully lazy evaluation which we call *Fully Lazy Functional Machine* (FLFM) will be defined in Section 3. Rules for compiling FLL programs to generate FLFM instructions will be described in Section 4. And an implementation method of FLFM with code generation for a conventional computer will be explained in Section 5.

2. Lispkit Lisp and Fully Lazy Lisp

Lispkit Lisp is a small functional language of which syntax is borrowed from Lisp and semantics is, however, completely different from that of Lisp. Lispkit Lisp is purely functional and excludes side-effects such as *setq* in Lisp. A recent version of Lispkit Lisp adopts lazy evaluation semantics as standard and provides a special mechanism for eager evaluation. Many programs including a compiler and a text editor have been written and published [6]. We have chosen Lispkit Lisp as our source language to explore the applicability of full laziness in practical functional programming.

2.1. Lispkit Lisp

Let e_0, e_1, $\cdots$ stand for Lispkit Lisp expressions, and x_1, x_2, $\cdots$ for variables. Then, a Lispkit Lisp expression e is either

(LK1) an integer n, or a symbol s representing itself,

(LK2) (*quote* e_0) of which value is the expression e_0,

(LK3) $(e_0\, e_1 \cdots e_n)$, which is an application of (curried) function e_0 to arguments e_1, $\cdots$, e_n.

(LK4) (*lambda* $(x_1 \cdots x_k)\, e_0$) for lambda expression

$$\lambda x_1 \cdots x_k .\, e_0$$

(LK5) (*let* $e_0\, (x_1 \,.\, e_1) \cdots (x_m \,.\, e_m)$) for declaration

$$e_0 \text{ where } x_1 = e_1$$
$$\text{and } \cdots \text{ and } x_m = e_m$$

or

(LK6) (*letrec* $e_0\ (x_1 . e_1)\ \cdots\ (x_m . e_m)$) for recursive declaration

$$e_0 \ \textbf{whererec}\ x_1 = e_1 \ \textbf{and} \cdots \textbf{and}\ x_m = e_m \ .$$

2.2. Fully Lazy Lisp

The target language of translation is a yet another Lisp called *Fully Lazy Lisp* (FLL). Translating Lispkit Lisp programs into FLL programs aims at implementing full laziness by ordinary lazy evaluation of FLL.

Fully Lazy Lisp is defined as follows: Let e_0, e_1, $\cdots$ stand for FLL expressions, and x_1, x_2, $\cdots$, y_1, y_2, $\cdots$ for variables. An FLL expression is either

(FLL1) an integer n, or a symbol s,

(FLL2) (*quote* e_0),

(FLL3) ($e_0\ e_1\ \cdots\ e_n$),

(FLL4) (*lam* ($y_1\ \cdots\ y_k$) e_0),

or

(FLL5) (*lam** ($y_1\ \cdots\ y_k$) e_0 $(x_1 . e_1)\ \cdots\ (x_m . e_m)$)

2.3. Translation of Lispkit Lisp into Fully Lazy Lisp

The translation algorithm, which we call *lambda-hoisting*, of Lispkit Lisp into FLL is an extension of Hughes' algorithm for finding *super-combinators* of applicative expressions [7].

3. Fully Lazy Functional Machine

In this section we will first explain design principles of our machine model, called ***Fully Lazy Functional Machine*** (FLFM), for evaluating FLL programs. It can be considered as a variant of the ***SECD*** machine [5,10] specifically designed for our purpose. Then we will specify the rules for compiling FLL programs into FLFM code.

3.1. Design Principles

(P1) Make full use of the linear environment structure.

(P2) Closure structure should be constructed with little effort.

(P3) Partially parametrized function should be represented as a sharable value, and the environment should have a sharable structure.

3.2. Machine Structure

The FLFM machine consists of four registers S, E, C, and D, each of which holds a list representing the *stack*, the *environment*, the *control* code, or the *dump*, respectively. It should be noted that the SECD model of FLFM is a conceptual one; the stack need not be of the list structure in actual implementation, for example. We will discuss about implementation detail in later sections.

We will follow the notation used in [5] to specify the machine by state transition as

$$S \quad E \quad C \quad D \ \rightarrow\ S' \quad E' \quad C' \quad D'$$

To describe the change of the environment E, we will use the convention that E_i means the i-th link of E, and $*E_i$ the contents, or the value, of the i-th element of E. Note that arguments passed to functions are moved from the stack S to the environment so that their values are to be referenced as $*E_i$, not as E_i. We will denote a closure consisting of code C and environment E by $[C{:}E]$, and an empty list by ϕ instead of **nil**.

3.2.1. Load Instructions

Load Constant

$$\begin{array}{l} S \quad E \quad (\,LD\_SEXP\ x\ .\ C\,) \quad D \\ \quad \rightarrow\ (\,x\ .\ S\,) \quad E \quad C \quad D \end{array}$$

where x is a quoted S-expression.

Load Combinator

$$\begin{array}{l} S \quad E \quad (\,LD\_COMB\ x\ .\ C\,) \quad D \\ \quad \rightarrow\ (\,[C'{:}\phi]\ .\ S\,) \quad E \quad C \quad D \end{array}$$

where x is a global combinator or an anonymous combinator of the form (FLL4) or (FLL5), and C' is the code for x. Combinator is loaded on the stack as a closure with empty environment.

Load Closure

$$\begin{array}{l} S \quad E \quad (\,LD\_CLOS\ C'\ .\ C\,) \quad D \\ \quad \rightarrow\ (\,[C'{:}E]\ .\ S\,) \quad E \quad C \quad D \end{array}$$

where C' stands for the code to be evaluated under environment E.

Load Argument

$$\begin{array}{l} S \quad E \quad (\,LD\_ARG\ i\ .\ C\,) \quad D \\ \quad \rightarrow\ (\,*E_i\ .\ S\,) \quad E \quad C \quad D \end{array}$$

3.2.2. Environment Control Instructions

Extend Environment

$$\begin{array}{l} (\,x\ .\ S\,) \quad E \quad (\,EXT\_ENV\ .\ C\,) \quad D \\ \quad \rightarrow\ S \quad (\,x\ .\ E\,) \quad C \quad D \end{array}$$

$$\begin{array}{l} \phi \quad E \quad (\,EXT\_ENV\ .\ C\,) \quad (\,S'\ E'\ C'\ .\ D\,) \\ \quad \rightarrow\ (\,[C''{:}E]\ .S'\,) \quad E' \quad C' \quad D \end{array}$$

where C'' stands for (EXT_ENV . C). The second rule shows how partially evaluated function is obtained.

Extend Recursive Environment

$$\begin{array}{l} S \quad E \quad (\,EXT\_RECENV\ C'\ .\ C\,) \quad D \\ \quad \rightarrow\ S \quad E' \quad C \quad D \end{array}$$

where E' points to the same cell as E, i.e., $E'=E$, with $*E_0$ being moved to newly created cell E'_1 and $*E'_0=[C'{:}E']$, $E'_{i+1}=E_i$ for $i \geq 1$. In case of $E=\phi$, E' is obtained by simply extending E using a new cell with

$*E'_0=[C':E']$. This instruction effectively creates circular structures for recursive declaration.

3.2.3. Evaluation and Application Instructions

Evaluate

$$\begin{array}{l}(x\,.\,S)\quad E\quad (EVAL\,.\,C)\quad D\\ \qquad\rightarrow\ (x\,.\,S)\quad E\quad C\quad D\end{array}$$

where x is not a closure.

$$\begin{array}{l}([\phi:x]\,.\,S)\quad E\quad (EVAL\,.\,C)\quad D\\ \qquad\rightarrow\ (x\,.\,S)\quad E\quad C\quad D\end{array}$$

where $[\phi:x]$ is an indirection closure described below.

$$\begin{array}{l}([C':E']\,.\,S)\quad E\quad (EVAL\,.\,C)\quad D\\ \qquad\rightarrow\ \phi\quad E'\quad C'\quad (S\ E\ C\,.\,D)\end{array}$$

Apply

$$\begin{array}{l}(x\,.\,\phi)\quad E\quad \phi\quad (S'\ E'\ C'\,.\,D)\\ \qquad\rightarrow\ (x\,.\,S')\quad E'\quad C'\quad D\end{array}$$

where x is not a closure.

$$\begin{array}{l}([C':E']\,.\,S)\quad E\quad \phi\quad D\\ \qquad\rightarrow\ S\quad E'\quad C'\quad D\end{array}$$

This instruction, actually the end of code sequence, causes return to the caller of recursive evaluation if the element at the top of S is not a closure, or otherwise applies the closure to arguments on S.

Update

$$\begin{array}{l}(x\,.\,S)\quad E\quad (UPDATE\ i\,.\,C)\quad D\\ \qquad\rightarrow\ (x\,.\,S)\quad E'\quad C\quad D\end{array}$$

where $E'=E$ and $E'_j=E_j$ for every j; when $*E_i$ is a closure, contents of the cell pointed to by it is changed to an *indirection closure* with its code part ϕ and environment part x. In all cases, $*E_i$ is assigned x. The indirection closure is used to realize full laziness in FLFM.

The second case of *EVAL* instruction deals with evaluation of the indirection closure. Garbage collector can eliminate indirection closures in actual implementation.

3.3. Primitive Functions

There are several primitive functions in Lispkit Lisp and FLL. They are actually combinators in the sense that they are closed and do not have any free variables. In this section, we will show how these primitive functions can be implemented by a small set of FLFM instructions. In later sections, we will discuss about some optimization rules to gain efficiency.

3.3.1. Arithmetic and Boolean Operations

We first consider the combinator *add* for addition of two integers. Assume that we have an instruction *ADD* which adds two elements on the stack S and puts the result on the top of S.

```
add =
  ( EXT_ENV ; EXT_ENV ;
    LD_ARG 0 ; EVAL ; UPDATE 0 ;
    LD_ARG 1 ; EVAL ; UPDATE 1 ;
  ADD )
```

Other arithmetic operations such as *sub*, *mul*, etc., and Boolean operations as *eq* are defined quite similarly.

3.3.2. List Operations

Primitive functions for the list structure differ a bit. The constructor *cons* for list cells should not evaluate arguments in lazy evaluation [5].

```
cons =
  ( EXT_ENV ; EXT_ENV ;
    LD_ARG 0 ; LD_ARG 1 ;
  CONS )
```

$$\begin{array}{l}(x\ y\,.\,S)\quad E\quad (CONS\,.\,C)\quad D\\ \qquad\rightarrow\ ((x.y)\,.\,S)\quad E\quad C\quad D\end{array}$$

The selectors *head* and *tail* need to evaluate the argument and take an appropriate component of the pair.

```
head =
  ( EXT_ENV ;
    LD_ARG 0 ; EVAL ; UPDATE 0 ;
  CAR )
```

```
tail =
  ( EXT_ENV ;
    LD_ARG 0 ; EVAL ; UPDATE 0 ;
  CDR )
```

where *CAR* and *CDR* are FLFM instructions.

$$\begin{array}{l}((x.y)\,.\,S)\quad E\quad (CAR\,.\,C)\quad D\\ \qquad\rightarrow\ (x'\,.\,S)\quad E\quad C\quad D\end{array}$$

$$\begin{array}{l}((x.y)\,.\,S)\quad E\quad (CDR\,.\,C)\quad D\\ \qquad\rightarrow\ (y'\,.\,S)\quad E\quad C\quad D\end{array}$$

where x' and y' are evaluated components of $(x.y)$.

The predicates *atom* and *null* over list structures are defined similarly.

3.3.3. Conditional Operation

The combinator *if* can be written using a conditional instruction *IF* as

```
if =
  ( EXT_ENV ;
    LD_ARG 0 ; EVAL ; UPDATE 0 ;
  IF )
```

The instruction *IF* selects either *if_true* or *if_false* according to the value at the top of the stack:

$$\begin{array}{l}(\mathbf{true}.S)\quad E\quad (IF)\quad D\\ \qquad\rightarrow\ ([if\_true:\phi].S)\quad E\quad \phi\quad D\end{array}$$

$$\begin{array}{l}(\mathbf{false}.S)\quad E\quad (IF)\quad D\\ \qquad\rightarrow\ ([if\_false:\phi].S)\quad E\quad \phi\quad D\end{array}$$

The combinators *if_true* and *if_false* are defined as

$$\begin{array}{l} \mathit{if\_true} = \\ \quad (\ \mathit{EXT\_ENV}\ ;\ \mathit{EXT\_ENV}\ ; \\ \qquad \mathit{LD\_ARG}\ 1\ ;\ \mathit{EVAL}\ ;\ \mathit{UPDATE}\ 1\ ) \end{array}$$

$$\begin{array}{l} \mathit{if\_false} = \\ \quad (\ \mathit{EXT\_ENV}\ ;\ \mathit{EXT\_ENV}\ ; \\ \qquad \mathit{LD\_ARG}\ 0\ ;\ \mathit{EVAL}\ ;\ \mathit{UPDATE}\ 0\ ) \end{array}$$

4. Compilation of FLL into FLFM Code

Rules for compiling FLL expressions into FLFM code is simpler than that for compiling Lispkit Lisp expressions into ordinary SECD machine.

4.1. Compilation Rules

Let

$$\rho = [u_0, u_1, \cdots, u_p]$$

stand for the *static environment* to lookup variables in lexical-addressing. We will use the notation in [5]:

$$e * \rho$$

represents FLFM code for FLL expression e with respect to the environment ρ, and

$$(\ s_1\ ) \mid (\ s_2\ ) \mid \cdots \mid (\ s_n\ )$$

represents

$$(\ s_1\ s_2\ \cdots\ s_n\ )$$

And we will use curly braces { } to group code sequences, and $\mathit{EXT\_ENV}^k$ to represent a sequence of *EXT_ENV* instruction of length k. The basic compilation rules can be described as follows.

(C1) Integer n

$$n*\rho = (\mathit{LD\_COMB}\ n)$$

Symbol s

$$s*[u_0, u_1, \cdots, u_p] = \begin{cases} (\mathit{LD\_ARG}\ i) & \\ \qquad \text{if } e = u_i \text{ in } \rho \text{ for some } i & \\ (\mathit{LD\_COMB}\ s) & \text{otherwise} \end{cases}$$

(C2) Quotation

$$(\mathit{quote}\ e_0)*\rho = (\mathit{LD\_SEXP}\ e_0)$$

(C3) Applicative form

$$\begin{array}{l} (e_0\ e_1\ \cdots\ e_n)*\rho = \\ \quad (\mathit{LD\_CLOS}\ \{\ e_n*\rho \mid \cdots \mid e_1*\rho \mid e_0\bullet\rho\ \}\ ) \end{array}$$

(C4) *lam* combinator

$$\begin{array}{l} (\mathit{lam}(y_1 \cdots y_k)e_0)*\rho = \\ \quad (\mathit{LD\_COMB}\ \{\ (\ \mathit{EXT\_ENV}^k\ ) \mid e_0\bullet\rho'\ \}\ ) \end{array}$$

where $\rho' = [y_k, \cdots, y_1]$.

(C5) *lam** combinator

$$\begin{array}{l} (\mathit{lam}^*(y_1 \cdots y_k)e_0(x_1.e_1) \cdots (x_m.e_m))*\rho = \\ \quad (\mathit{LD\_COMB}\ \{\ (\ \mathit{EXT\_ENV}^k\ ) \\ \qquad \mid (\mathit{EXT\_RECENV}\ e_1*\rho') \mid \\ \qquad\quad \cdots \mid (\mathit{EXT\_RECENV}\ e_m*\rho') \mid e_0\bullet\rho'\ \}\ ) \end{array}$$

where $\rho' = [x_m, \cdots, x_1, y_k, \cdots, y_1]$.

(C6) Auxiliary rule

$$e\bullet\rho = \begin{cases} (\mathit{LD\_ARG}\ i) \mid (\mathit{EVAL}) \mid (\mathit{UPDATE}\ i) & \\ \qquad \text{if } e = u_i \text{ in } \rho \text{ for some } i & \\ e*\rho \mid (\mathit{EVAL}) & \text{otherwise} \end{cases}$$

where $\rho = [u_0, u_1, \cdots, u_p]$.

Note that $e\bullet\rho$ represents a code sequence to evaluate the expression e. If e is an argument held in the environment, it should be replaced by the result of evaluation. In the general compilation scheme, the head term of the applicative form is forced to be evaluated.

4.2. Optimization

The compilation rules described above do not use any specific information about primitive functions. If we had used such information, we could obtain better FLFM code. Therefore, we will discuss some rules for optimization in this section. In doing so, we need to introduce a few FLFM instructions to gain efficiency.

4.2.1. Update Operation

The first rule we consider is for the code sequence

$$\mathit{LD\_ARG}\ i\ ;\ \mathit{EVAL}\ ;\ \mathit{UPDATE}\ i$$

which is generated for an argument at the front of the applicative form. Introducing a new FLFM instruction *LD_ARG_EVAL* eliminates redundant operations to locate the i-th argument on the environment.

$$\begin{array}{l} S \quad E \quad (\mathit{LD\_ARG\_EVAL}\ i\ .\ C) \quad D \\ \qquad \rightarrow \ (x\ .\ S) \quad E \quad C \quad D \end{array}$$

if $*E_i$ is not a closure, or

$$\begin{array}{l} S \quad E \quad (\mathit{LD\_ARG\_EVAL}\ i\ .\ C) \quad D \\ \qquad \rightarrow \ \phi \quad E' \quad C' \quad (S\ E_i\ (\mathit{UPD})\ E\ C\ .\ D) \end{array}$$

if $*E_i = [C' : E']$.

The instruction *UPD* is never generated by the compiler.

$$\begin{array}{l} (x\ .S) \quad E_i \quad (\mathit{UPD}) \quad (E\ C\ .\ D) \\ \qquad \rightarrow \ (x\ .\ S) \quad E' \quad C \quad D \end{array}$$

with $*E_i := x$ as shown in the rule for *UPDATE*.

4.2.2. Evaluate Operation

For a code sequence

$$\mathit{LD\_CLOS}\ C'\ ;\ \mathit{EVAL}$$

it is observed that

$$\begin{array}{l} S \quad E \quad (\mathit{LD\_CLOS}\ C'\ \mathit{EVAL}\ .\ C) \quad D \\ \rightarrow \quad ([C':E]\ .\ S) \quad E \quad (\mathit{EVAL}\ .\ C) \quad D \\ \rightarrow \quad \phi \quad E \quad C' \quad (S\ E\ C\ .\ D) \end{array}$$

Thus, no closure is required if a new instruction *LD_CLOS_EVAL* is introduced.

$$\begin{array}{l} S \quad E \quad (\mathit{LD\_CLOS\_EVAL}\ C'\ .\ C) \quad D \\ \quad \rightarrow \quad \phi \quad E \quad C' \quad (S\ E\ C\ .\ D) \end{array}$$

Moreover, it should be noted that the compilation rules allow to write

$$((e'_0\, e'_1 \cdots e'_m)\, e_1 \cdots e_n)^{*}\rho$$

$$= (\mathit{LD\_CLOS}\ \{\ e_n{}^{*}\rho \mid \cdots \mid e_1{}^{*}\rho \mid e'_m{}^{*}\rho \mid e'_1{}^{*}\rho \mid \cdots \mid e'_0{\bullet}\rho\ \})$$

This corresponds to the fact that

$$((e'_0\, e'_1 \cdots e'_m)\, e_1 \cdots e_n)$$

is semantically equivalent to a simpler form

$$(e'_0\, e'_1 \cdots e'_m\, e_1 \cdots e_n)$$

Therefore, any applicative expression can be repeatedly transformed into a simpler form until the head term is either a combinator or an argument.

Similar simplification can be applied to the case where an anonymous combinator appears at the head of the applicative form. We can see that

$$\begin{array}{l} S \quad E \quad (\mathit{LD\_COMB}\ C'\ \mathit{EVAL}) \quad D \\ \rightarrow \quad ([C':\phi]\ .\ S) \quad E \quad (\mathit{EVAL}) \quad D \\ \rightarrow \quad \phi \quad \phi \quad C' \quad (S\ E\ \phi\ .\ D) \\ \rightarrow \quad ([C':\phi]\ .\ S) \quad E \quad \phi \quad D \\ \rightarrow \quad S \quad \phi \quad C' \quad D \end{array}$$

Note that C' begins with *EXT_ENV* and evaluation with empty stack results in immediate return with a closure. Thus, the *EVAL* instruction following *LD_COMB* is redundant and may be omitted; that is,

$$\mathit{LD\_COMB}\ C'\ ;\mathit{EVAL}$$

can always be simplified to

$$\mathit{LD\_COMB}\ C' \ .$$

If a new instruction, *CALL*, is introduced for applying combinators, closures are not created by replacing (*LD_COMB C' EVAL*) with (*CALL C'*).

$$\begin{array}{l} S \quad E \quad (\mathit{CALL}\ C') \quad D \\ \quad \rightarrow \quad S \quad \phi \quad C' \quad D \end{array}$$

From the above discussion, we can rewrite the compilation rule (C6) as

(C6') Auxiliary rule

$$e{\bullet}\rho = \begin{cases} (\mathit{LD\_ARG\_EVAL}\ i) & \\ \qquad \text{if } s = u_i \text{ in } \rho \text{ for some } i & \\ e^{*}\rho \quad \text{if } e \text{ is a quotation or a combinator} & \\ e^{*}\rho \mid (\mathit{EVAL}) \quad \text{otherwise} & \end{cases}$$

where $\rho = [u_0, u_1, \cdots, u_p]$. When $e{\bullet}\rho$ appears at the head of an applicative form, further optimization can be taken as described above.

4.2.3. Primitive Operations

Suppose that we have a term (*add* $e_1\ e_2$). If we use the knowledge about the arity of *add*, the environment consisting of e_1 and e_2 is not necessary. In such a case, we can generate FLFM code as

$$(\mathit{add}\ e_1\ e_2)^{*}\rho = \{\ e_2{\bullet}\rho \mid e_1{\bullet}\rho \mid (\mathit{ADD})\ \} \ .$$

Similar rules can be applied to other arithmetic and Boolean operations.

For the list constructor *cons*, and for the selectors *head* and *tail*, we have

$$(\mathit{cons}\ e_1\ e_2)^{*}\rho = \{\ e_2{}^{*}\rho \mid e_1{}^{*}\rho \mid (\mathit{CONS})\ \}$$

$$(\mathit{head}\ e_1)^{*}\rho = \{\ e_1{\bullet}\rho \mid (\mathit{CAR})\ \}$$

$$(\mathit{tail}\ e_1)^{*}\rho = \{\ e_1{\bullet}\rho \mid (\mathit{CDR})\ \}$$

Given the conditional form (*if* $e_1\ e_2\ e_3$), we can optimize the term when three arguments are supplied together. Recall that

$$(\mathit{if}\ e_1)^{*}\rho = \{\ e_1{\bullet}\rho \mid (\mathit{IF})\ \}$$

and *IF* yields a combinator *if_true* or *if_false*, which in turn selects e_2 or e_3 to be evaluated. A simple way to optimize the code for the above form might be to provide FLFM instructions for directly evaluating alternatives. However, these instructions fail to update the values of e_2 and e_3 given as arguments. Our solution to this problem is to introduce an FLFM instruction *SELECT*

$$\begin{array}{l} (\mathbf{true}\ .\ S) \quad E \quad (\mathit{SELECT}\ C_2\ C_3) \quad D \\ \quad \rightarrow \quad S \quad E \quad C_2 \quad D \end{array}$$

$$\begin{array}{l} (\mathbf{false}\ .\ S) \quad E \quad (\mathit{SELECT}\ C_2\ C_3) \quad D \\ \quad \rightarrow \quad S \quad E \quad C_3 \quad D \end{array}$$

and to compile the term as

$$\begin{array}{l} (\mathit{if}\ e_1\ e_2\ e_3)^{*}\rho = \\ \quad \{\ e_1{\bullet}\rho \mid (\mathit{SELECT}) \mid e_2{\bullet}\rho \mid e_3{\bullet}\rho\ \} \end{array}$$

5. Implementation of FLFM

In this section, we will look over an FLFM implementation on a conventional machine MC68000.

As mentioned in Section 3.2, there is no need to use the list structure to represent every object held by the registers S, E, C, and D. In the first place, the stack S can be implemented by usual hardware stack manipulated by auto-increment and -decrement addressing of MC68000. The code C is a fixed code of MC68000 instructions, and controlled by the program counter. The dump D can be embedded in the stack using the frame pointer indicating stack frames for recursive activations of functions. To attain the sharing property (P3) of the environment E, it is reasonable to make the environment using the list structure.

Each value is represented by a 32 bit word of which first 8 bit byte is used for the *tag* part indicating value types. Remaining 24 bit field contains unboxed (integer, character, or Boolean) value, or a pointer to a cell allocated in the *heap* store. Four of the address registers of MC68000 are devoted to maintaining S, E, D, and the heap store. Figure 1 illustrates the stack, the heap, and pointers. Tags are not shown in the figure.

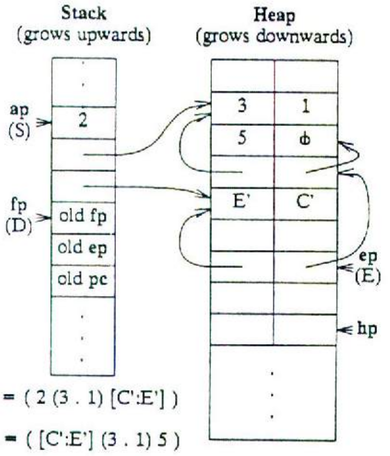

Figure 1

Most of the FLFM instructions are expanded using macro facilities of the assembler. And commonly used instruction sequences like *ext_env*, *apply*, etc., are supplied as run-time routines. An example of MC68000 code is shown in the Appendix. As for the performance, the code thus obtained runs as fast as about 3000 function calls per second on 8MHz MC68000 machines. This exceeds the execution count, about 2000, by a micro-coded interpreter of the SECD machine for Lispkit Lisp on the Perq [6].

We shall leave further implementation details and experimental results to [13].

6. Conclusion

Our primary concern in this paper is to develop compilation technique for fully lazy evaluation on conventional machines. Once we have obtained expressions of the Fully Lazy Lisp form, we can evaluate them in fully lazy way using ordinary lazy evaluation mechanism provided that it allows function application of insufficient number of arguments. Our FLFM has been designed as a basic model of fully lazy evaluators. As shown in the section on actual implementation, code generation from FLFM code turns out to be much easier than from ordinary SECD code. This enables us to enhance portability of compiler systems.

Johnsson [8] deals with a compilation scheme for ordinary lazy evaluation. It is, however, different from ours in several respects. Among others, it does not support full laziness, as is the basis of our method.

Although an advantageous feature of full laziness in practical problems has been demonstrated in [12], further investigation should be necessary. We believe that our compilation technique is extremely useful for exploiting the applicability of full laziness.

References

1. Aho, A.V., and Ullman, J.D.: *Principles of Compiler Design*. Addison-Wesley, 1977.
2. Burge, W.H.: *Recursive Programming Techniques*. Addison-Wesley, 1975
3. Friedman, D.P., and Wise, D.S.: Cons should not evaluate its arguments. *Automata, Languages and Programming*, 257-284. Edinburgh University Press, 1976.
4. Henderson, P., and Morris, J.M.: A lazy evaluator. *Proc. 3rd Symp. on Principles of Programming Languages*, 95-103 (1976)
5. Henderson, P.: *Functional Programming: Application and Implementation*. Prentice-Hall, 1980.
6. Henderson, P., Jones, G.A., and Jones, S.B.: *The Lispkit Manual*. Technical Monograph PRG-32, Oxford University Computing Laboratory, 1983.
7. Hughes, R.J.M.: Super-combinators: a new implementation method for applicative languages. *Proc. 1982 ACM Symp. Lisp and Functional Programming*, 1-10 (1982)
8. Johnsson, T.: Efficient compilation of lazy evaluation. *Proc. SIGPLAN '84 Symp. on Compiler Construction*, 58-69 (1984)
9. Johnsson, T.: *Lambda Lifting*. Programming Methodology Group Memo, Chalmers University of Technology, Goteborg, 1985.
10. Landin, P.J.: The next 700 programming languages. *Comm. ACM*, 9, 157-164 (1966)
11. Takeichi, M.: Evaluation of combinator expressions. *Proc. 1st JSSST Annual Conference*, 213-222 (1984). In Japanese.
12. Takeichi, M.: *Partial Parametrization Eliminates Multiple Traversals of Data Structures*. Technical Memo, Department of Computer Science, The University of Electro-Communications, 1985.
13. Takeichi, M.: *Implementation of Fully Lazy Functional Machine*. In preparation.

Appendix

Lispkit Lisp Source

```
(letrec filter
      (filter lambda (p)
            (lambda (l)
                  (if (p (head l))
                      (cons (head l) (filter p (tail l)))
                      (filter p (tail l))))
            ))
```

Fully Lazy Lisp Source

```
((lam* ( ) filter
     (filter (lam (f p)
               ((lam (r g l)
                  (if (r (head l))
                      (cons (head l) (g (tail l)))
                      (g (tail l))))
                 (p)
                 (f p)))
             (filter))))
```

FLFM code

```
$0:    CALL $1                  ;($1)
$1:    EXT_RECENV $2            ;(lam*()filter(filter.$2))

       LD_ARG_EVAL 0

       APPLY
$2:    LD_ARG 0                 ;($3 filter)
       CALL $3
$3:    EXT_ENV                  ;(lam(f p)($5 p $4))
       EXT_ENV
       LD_CLOS $4



       LD_ARG 0
       CALL $5

$4:    LD_ARG 0                 ;(f p)
       LD_ARG_EVAL 1


       APPLY
$5:    EXT_ENV                  ;(lam(r g l)(if $8 $7 $6))
       EXT_ENV
       EXT_ENV
       LD_CLOS_EVAL $8

       SELECT $7 $6

$6:    LD_CLOS $9               ;(g $9)



       LD_ARG_EVAL 1


       APPLY
$7:    LD_CLOS $6               ;(cons $10 $6)



       LD_CLOS $10



       CONS
$8:    LD_CLOS $10              ;(r $10)



       LD_ARG_EVAL 2



       APPLY
$9:    LD_ARG_EVAL 0            ;(tail l)

       CDR
$10:   LD_ARG_EVAL 0            ;(head l)

       CAR
```

MC68000 code

```
$0:    suba.l     ep,ep
$1:    lea        $2,a0
       bsr        ext_recenv
       move.l     ep,a0
       bsr        ld_arg_eval
       bra        apply
$2:    move.l     4(ep),-(ap)
       suba.l     ep,ep
$3:    bsr        ext_env
       bsr        ext_env
       move.l     hp,-(ap)
       lea        $4,a0
       move.l     a0,(hp)+
       move.l     ep,(hp)+
       move.l     4(ep),-(ap)
       suba.l     ep,ep
       bra        $5
$4:    move.l     4(ep),-(ap)
       move.l     ep,a0
       move.l     (a0),a0
       bsr        ld_arg_eval
       bra        apply
$5:    bsr        ext_env
       bsr        ext_env
       bsr        ext_env
       lea        $8,a0
       bsr        ld_clos_eval
       cmpi.l     #TRUE,(ap)+
       beq        $7
$6:    move.l     hp,-(ap)
       lea        $9,a0
       move.l     a0,(hp)+
       move.l     ep,(hp)+
       move.l     ep,a0
       move.l     (a0),a0
       bsr        ld_arg_eval
       bra        apply
$7:    move.l     hp,-(ap)
       lea        $6,a0
       move.l     a0,(hp)+
       move.l     ep,(hp)+
       move.l     hp,-(ap)
       lea        $10,a0
       move.l     a0,(hp)+
       move.l     ep,(hp)+
       bra        cons
$8:    move.l     hp,-(ap)
       lea        $10,a0
       move.l     a0,(hp)+
       move.l     ep,(hp)+
       move.l     ep,a0
       move.l     (a0),a0
       move.l     (a0),a0
       bsr        ld_arg_eval
       bra        apply
$9:    move.l     ep,a0
       bsr        ld_arg_eval
       bra        cdr
$10:   move.l     ep,a0
       bsr        ld_arg_eval
       bra        car
```