New Generation Computing, 5(1988) 377-391
OHMSHA, LTD. and Springer-Verlag

Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs

Masato TAKEICHI
Educational Computer Centre,
University of Tokyo,
2-11-16 Yayoi, Bunkyo-ku, Tokyo 113, Japan.

Received 24 September 1986
Revised manuscript received 10 November 1987

Abstract Lambda-hoisting is a technique for transforming functional programs into ones suitable for fully lazy evaluation. The proposed method has a great advantage in generating efficient code for conventional computers. The basic idea of lambda-hoisting is described with remarks on similar techniques, and a simple algorithm is presented in a formal way.

Keywords: Functional programming, Fully lazy evaluation, Program transformation

§1 Introduction

Compilation techniques for lazy functional languages have been studied by many researchers. Turner[16,17] proposes a novel scheme of generating *combinator expressions* as object code, and applies it to implement several functional languages.[15,18,19] Programs are compiled into expressions consisting of only pre-defined combinators. Hughes[5,6] generalizes this idea to generate code using *super-combinators* that are dependent on the source program and produced during compilation. On the other hand, modified versions of the classical *SECD machine*[3,10] adapted for lazy evaluation[1,2] are used as well.[4] The combinator code, including that using super-combinators, is usually represented by a graph and evaluation is taken by an interpreter that performs graph reduction. Such an evaluation method differs from the way the code of usual compiler languages runs to evaluate expressions. It is possible, however, to generate fixed code for conventional computers that evaluates combinator expressions.[9,11,12] The fixed program thus obtained can be considered as a program coded for a lazy SECD machine. Hence, there is no essential difference between two approaches at least in regard to lazy evaluation, though they seem

quite different at first sight.

However, there remains a great difference. The combinator approach enjoys *full laziness* in its nature. Full laziness is the property that every expression is evaluated at most once after the variables in it have been bound.[6] This property is not observed in the compiler for a lazy SECD machine[4] or in the *lambda-lifting* algorithm[7,8] for generating conventional machine code.

The method described in this paper is a departure from the super-combinator approach. The idea of the super-combinator[5,6] is based on

- extracting maximal free occurrences of expression of every lambda expres-

Syntactic domains

$b \in$ **Bas**	basic values
$x \in$ **Ide**	identifiers
$e \in$ **Exp**	expressions

Abstract syntax

$e ::= b \mid x \mid e\ e \mid$ **fn** x: $e \mid$
$\quad e$ **where** $x = e$ **and** ... **and** $x = e \mid$
$\quad e$ **whererec** $x = e$ **and** ... **and** $x = e$

Semantic domains

B	basic values
$\mathbf{E} = [\mathbf{B} + \mathbf{F}]_{\perp}$	expressible values
$\mathbf{F} = \mathbf{D} \to \mathbf{E}$	functions
$\mathbf{D} = \mathbf{E}$	denotable values
$\mathbf{U} = \mathbf{Ide} \to \mathbf{D}_{\perp}$	environments

Semantic functions

B: **Bas** $\to$ **B**	(unspecified)
E: **Exp** $\to$ **U** $\to$ **E**	

$E[\![b]\!]\rho = B[\![b]\!]$
$E[\![x]\!]\rho = \rho[\![x]\!]$
$E[\![e_0\ e_1]\!]\rho = (E[\![e_0]\!]\rho)(E[\![e_1]\!]\rho)$
$E[\![$**fn** x: $e_0]\!]\rho = \lambda\delta.E[\![e_0]\!](\rho + <x \to \delta>)$
$E[\![e_0$ **where** $x_1 = e_1$ **and** ... **and** $x_n = e_n]\!]\rho = E[\![e_0]\!]\rho'$
$\quad$ where $\rho' = \rho + \langle x_1 \to E[\![e_1]\!]\rho\rangle + \ \ldots\ + \langle x_n \to E[\![e_n]\!]\rho\rangle$
$E[\![e_0$ **whererec** $x_1 = e_1$ **and** ... **and** $x_n = e_n]\!]\rho = E[\![e_0]\!]\rho'$
$\quad$ where $\rho' = \rho + \langle x_1 \to E[\![e_1]\!]\rho'\rangle + \ \ldots\ + \langle x_n \to E[\![e_n]\!]\rho'\rangle$

Notations

Domain construction operator + stands for the disjoint sum.
For any domain **X**, $\mathbf{X}_{\perp} = \mathbf{X} + \{\mathbf{err}\}$.
For an environment ρ, $\rho + \langle x \to \delta\rangle$ denotes
$\quad \lambda y.$**if** $x = y$ **then** δ **else** $\rho[\![y]\!]$.

Initial Environment

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

Fig. 1 Denotational specification of a simple functional language.

sion as arguments of a global function, i.e., a super-combinator,
- defining the global function with coresponding number of parameters, i.e., the number of maximal free expressions plus the number of lambda variables in the original lambda expression, and
- choosing the parameter order so that the total number of parameters of the super-combinators for a program should be as few as possible.

It should be noted that the transformation based only on the first two rules generates super-combinators that share the best property of full laziness. The last principle is provided for efficiency. The number of arguments does nevertheless increase as the nesting level of lambda expressions becomes deeper. As the combinator reducer uses the stack to pass the arguments to combinators, many operations on the stack should be taken in reduction of super-combinators and they may cause the loss of efficiency.

To reduce the cost of parameter passing operations, we define a restricted class of applicative expressions, which we call *fully lazy normal form*. The fully lazy normal form is more general than the form of super-combinators in the sense that the latter is a special instance of the former. Evaluation of the program of the fully lazy normal form in an ordinary lazy way[1,2] results in fully lazy evaluation of the original program. In other words, it enables us to implement full laziness by means of an ordinary lazy evaluator. This is our basic strategy to achieve full laziness.

In this paper we present a technique called *lambda-hoisting* for transforming any program into fully lazy normal form. We use a simple functional language shown in Fig. 1.

§2 Fully Lazy Evaluation

In the fully lazy scheme, every expression is evaluated at most once, while only every argument of a function is evaluated at most once in ordinary lazy evaluation with a call-by-need[21] or a call-by-delayed-value[20] mechanism. We introduce here our basic idea of *lambda-hoisting* using an example by which Hughes[5] explains the motivation of using the super-combinator. The function *el* selects the n-th element of a linear list s*:

$$el = \textbf{fn}\ n{:}\{\textbf{fn}\ s{:}IF(=\ n\ 1)(HEAD\ s)(el(-\ n\ 1)(TAIL\ s))\}$$

The function *snd* that gives the second element of a list can be defined by instantiating this function with the first argument:

$$snd = el\ 2$$

* Functions written in upper case letters as *IF*, *HEAD* and *TAIL*, and prefix operators like = and − are taken as elements of **Ide**. These are pre-defined functions defined in the initial environment ρ_0. We do not specify the concrete syntax of the language in this paper; we insert symbols {and} or indent **where**-clauses to clarify textual scope.

Evaluation of the right hand side proceeds as*

$$
\begin{array}{l}
el\ 2 \rightarrow \\
\quad \textbf{fn}\ s{:}\{IF(=\nu_1 1)(HEAD\ s)(el(-\nu_1 1)(TAIL\ s))\ \textbf{where}\ \nu_1 = 2\}
\end{array}
$$

No more computation proceeds unless the argument for s is supplied. Assume that a list s_1 is given:

$$
\begin{array}{l}
snd\ s_1 \\
\quad \rightarrow IF(=\nu_1 1)(HEAD\ \sigma_1)(el(-\nu_1 1)(TAIL\ \sigma_1)) \\
\qquad \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow IF\ \textbf{false}(HEAD\ \sigma_1)(el(-\nu_1 1)(TAIL\ \sigma_1)) \\
\qquad \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow IF\text{-}FALSE(HEAD\ \sigma_1)(el(-\nu_1 1)(TAIL\ \sigma_1)) \\
\qquad \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow el(-\nu_1 1)(TAIL\ \sigma_1)\ \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow \{IF(=\nu_2 1)(HEAD\ \sigma_2)(el(-\nu_2 1)(TAIL\ \sigma_2)) \\
\qquad \textbf{where}\ \sigma_2 = TAIL\ \sigma_1\ \textbf{and}\ \nu_2 = (-\nu_1 1)\}\ \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow \{IF(=\nu_2 1)(HEAD\ \sigma_2)(el(-\nu_2 1)(TAIL\ \sigma_2)) \\
\qquad \textbf{where}\ \sigma_2 = TAIL\ \sigma_1\ \textbf{and}\ \nu_2 = 1\}\ \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow IF\ \textbf{true}(HEAD\ \sigma_2)(el(-\nu_2 1)(TAIL\ \sigma_2)) \\
\qquad \textbf{where}\ \sigma_2 = TAIL\ \sigma_1\ \textbf{and}\ \nu_2 = 1 \\
\qquad\qquad \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow IF\text{-}TRUE(HEAD\ \sigma_2)(el(-\nu_2 1)(TAIL\ \sigma_2)) \\
\qquad \textbf{where}\ \sigma_2 = TAIL\ \sigma_1 \textbf{and}\ \nu_2 = 1 \\
\qquad\qquad \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2 \\
\quad \rightarrow \{HEAD\ \sigma_2\ \textbf{where}\ \sigma_2 = TAIL\ \sigma_1\ \textbf{and}\ \nu_2 = 1\} \\
\qquad \textbf{where}\ \sigma_1 = s_1\ \textbf{and}\ \nu_1 = 2
\end{array}
$$

When the function *snd* is applied to another list s_2 after $(snd\ s_1)$ is evaluated, similar steps are necessarily taken. However, since the first argument of *el* is common to both $(snd\ s_1)$ and $(snd\ s_2)$, some part of computation is made to be done only once. To this end, a simple transformation of the original function works well; extracting all the terms dependent on the first parameter n and keeping their values for later use once they are evaluated. By rewriting the definition, we get

$$
\begin{array}{l}
el = \textbf{fn}\ n{:} \\
\quad \{\textbf{fn}\ s{:}\ a(HEAD\ s)(b(TAIL\ s)) \\
\qquad\qquad \textbf{whererec}\ a = IF(=n\ 1)\ \textbf{and}\ b = el(-n\ 1)\}
\end{array}
$$

Then we can define

* We shall write down the computational process using a similar notation as the source language; expressions bound to parameters are represented by **where** - or **whererec** - clauses with generated fresh variables.

$$
\begin{aligned}
&snd = el\ 2\\
&\rightarrow \textbf{fn}\ s\colon\ \alpha_1(HEAD\ s)(\beta_1(TAIL\ s))\\
&\quad \textbf{whererec}\ \alpha_1 = IF(=\nu_1 1)\ \textbf{and}\ \beta_1 = el(-\nu_1 1)\ \textbf{and}\ \nu_1 = 2
\end{aligned}
$$

Evaluation of $(snd\ s_1)$ proceeds as follows:

$$
\begin{aligned}
&snd\ s_1\\
&\quad \rightarrow \alpha_1(HEAD\ \sigma_1)(\beta_1(TAIL\ \sigma_1))\ \textbf{whererec}\ \sigma_1 = s_1\\
&\qquad \textbf{whererec}\ \alpha_1 = IF(=\nu_1 1)\ \textbf{and}\ \beta_1 = el(-\nu_1 1)\ \textbf{and}\ \nu_1 = 2\\
&\quad \rightarrow \alpha_1(HEAD\ \sigma_1)(\beta_1(TAIL\ \sigma_1))\ \textbf{whererec}\ \sigma_1 = s_1\\
&\qquad \textbf{whererec}\ \alpha_1 = IF\text{-}FALSE\ \textbf{and}\ \beta_1 = el(-\nu_1 1)\ \textbf{and}\ \nu_1 = 2\\
&\quad \rightarrow IF\text{-}FALSE(HEAD\ \sigma_1)(\beta_1(TAIL\ \sigma_1))\ \textbf{whererec}\ \sigma_1 = s_1\\
&\qquad \textbf{whererec}\ \alpha_1 = IF\text{-}FALSE\ \textbf{and}\ \beta_1 = el(-\nu_1 1)\ \textbf{and}\ \nu_1 = 2\\
&\quad \rightarrow \beta_1(TAIL\ \sigma_1)\ \textbf{whererec}\ \sigma_1 = s_1\\
&\qquad \textbf{whererec}\ \alpha_1 = IF\text{-}FALSE\ \textbf{and}\ \beta_1 = el(-\nu_1 1)\ \textbf{and}\ \nu_1 = 2
\end{aligned}
$$

The first term β_1 becomes

$$
\begin{aligned}
&\beta_1 = \textbf{fn}\ s\colon\ \alpha_2(HEAD\ s)(\beta_2(TAIL\ s))\\
&\qquad \textbf{whererec}\ \alpha_2 = IF(=\nu_2 1)\ \textbf{and}\ \beta_2 = el(-\nu_2 1)\ \textbf{and}\ \nu_2 = (-\nu_1 1)
\end{aligned}
$$

and

$$
\begin{aligned}
&\beta_1(TAIL\ \sigma_1)\\
&\quad \rightarrow \alpha_2(HEAD\ \sigma_2)(\beta_2(TAIL\ \sigma_2))\ \textbf{whererec}\ \sigma_2 = TAIL\ \sigma_1\\
&\qquad \textbf{whererec}\ \alpha_2 = IF(=\nu_2 1)\ \textbf{and}\ \beta_2 = el(-\nu_2 1)\ \textbf{and}\ \beta_2 = (-\nu_1 1)\\
&\quad \rightarrow \alpha_2(HEAD\ \sigma_2)(\beta_2 TAIL\ \sigma_2))\ \textbf{whererec}\ \sigma_2 = TAIL\ \sigma_1\\
&\qquad \textbf{whererec}\ \alpha_2 = IF(=\nu_2 1)\ \textbf{and}\ \beta_2 = el(-\nu_2 1)\ \textbf{and}\ \nu_2 = 1\\
&\quad \rightarrow \alpha_2(HEAD\ \sigma_2)(\beta_2(TAIL\ \sigma_2))\ \textbf{whererec}\ \sigma_2 = TAIL\ \sigma_1\\
&\qquad \textbf{whererec}\ \alpha_2 = IF\text{-}TRUE\ \textbf{and}\ \beta_2 = el(-\nu_2 1)\ \textbf{and}\ \nu_2 = 1\\
&\quad \rightarrow IF\text{-}TRUE(HEAD\ \sigma_2)(\beta_2(TAIL\ \sigma_2))\ \textbf{whererec}\ \sigma_2 = TAIL\ \sigma_1\\
&\qquad \textbf{whererec}\ \alpha_2 = IF\text{-}TRUE\ \textbf{and}\ \beta_2 = el(-\nu_2 1)\ \textbf{and}\ \nu_2 = 1\\
&\quad \rightarrow HEAD\ \sigma_2\ \textbf{whererec}\ \sigma_2 = TAIL\ \sigma_1\\
&\qquad \textbf{whererec}\ \alpha_2 = IF\text{-}TRUE\ \textbf{and}\ \beta_2 = el(-\nu_2 1)\ \textbf{and}\ \nu_2 = 1
\end{aligned}
$$

We have thus obtained the result of $(snd\ s_1)$. At the same time, we have a new version of the function *snd*:

$$
\begin{aligned}
&snd = \textbf{fn}\ s\colon\ \alpha_1(HEAD\ s)(\beta_1(TAIL\ s))\\
&\qquad \textbf{whererec}\ \alpha_1 = IF\text{-}FALSE\\
&\qquad \textbf{and}\ \beta_1 = \textbf{fn}\ s\colon\ \alpha_2(HEAD\ s)(\beta_2(TAIL\ s))\\
&\qquad\quad \textbf{whererec}\ \alpha_2 = IF\text{-}TRUE\ \textbf{and}\ \beta_2 = el(-\nu_2 1)\ \textbf{and}\ \nu_2 = 1\\
&\qquad \textbf{and}\ \nu_1 = 2
\end{aligned}
$$

All the terms dependent on n have already been evaluated. When $(snd\ s_2)$ is to be evaluated, only the necessary computational steps dependent on s_2 have to be taken. We have thus attained full laziness in evaluating both the terms $(snd\ s_1)$

and $(snd\ s_2)$ in a program.

We call the transformation technique just demonstrated *lambda-hoisting*. We shall deal with an algorithm for lambda-hoisting in the rest of the paper.

§3 Lambda-Hoisting Algorithm

As mentioned in Section 1, evaluation of the combinator expression achieves fully lazy evaluation of the source program. Several methods for implemention of the evaluator have been proposed. The combinators selected by Turner[16] are so primitive that many execution steps are needed to reduce the combinator expression. Hughes[5] improves this by generating super-combinators specifically chosen for the source program. It is reported that the super-combinator code runs several times faster than Turner's code.[6,11,12]

Lambda-hoisting is similar to the super-combinator method. For the function *el* in the previous section, we can derive a super-combinator ϕ by Hughes' method as*

$$el = \phi(IF\ (=\ n\ 1))(el(-\ n\ 1))$$
$$\textbf{where}\ \phi = \textbf{fn}\ a\ b\ s\colon a(HEAD\ s)(b(TAIL\ s))$$

The super-combinator is a closed function that does not contain any free variables in it and it can be treated as a global function. The transformation using super-combinators suffers the inefficiency caused by the increase in the number of operations for passing arguments of a super-combinator to other super-combinators. In addition to this problem, it remains open how to compile recursive definitions into what kind of combinators. For example, the function *el* might be expressed using the fixed-point combinator **Y** as

$$el = \mathbf{Y}(\textbf{fn}\ el\colon\ \phi(IF(=\ n\ 1))(el(-\ n\ 1)))$$

However, when more than one recursive functions are included, it becomes difficult to express the combinator code in a form from which the original recursive definition can be presumed. The program transformed by lambda-hoisting preserves recursive forms as they have been in the original program.

3.1 Fully Lazy Normal Form

The basic idea of lambda-hoisting is to transform source programs into programs consisting of a more general form of functions than super-combinators. The resulting function may contain local definitions in a restricted way, while the super-combinator must not. The *fully lazy normal form* is a sub-language of the language defined in Fig. 1. Its syntax rules and context condition are shown in Fig. 2.

The semantic function E of Fig. 1 is supposed to be applied to the new

* We shall use hereafter an extended form of **fn**-abstraction: **fn** $x_1 x_2 \ldots x_n : e_0$. It is a natural extension to **fn** $x : e_0$, while it is not equivalent to **fn** x_1: (**fn** x_2: ... (**fn** x_n: e_0))). See Section 3.3.

Syntax

$e ::= e' \mid e'$ **whererec** $x = e'$ **and** ... **and** $x = e'$
$e' ::= b \mid x \mid e'e' \mid$ **fn** x: e

Context condition

Expression e does not contain any free occurrences of compound expressions.

Fig. 2 Fully lazy normal form.

syntactic category e' as well as e. Note that **where**-clauses disappear in the fully lazy normal form. Local definitions by **whererec**-clauses may appear only in the outermost expression, i.e., the program, or in the body of **fn**-abstraction. That is, the transformed program is of either form

$$e_0$$

or

$$e_0 \textbf{ whererec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n$$

The expression in general is composed of primitive elements b from **Bas** and x from **Ide,** and functions of the form

$$\textbf{fn } x\text{: } e_0$$

or

$$\textbf{fn } x\text{: } e_0 \textbf{ whererec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n$$

As exemplified in the example of *el,* expressions e_1, ..., e_n in the last form are *maximal occurrences* of expressions dependent on x in the original **fn**-body, and e_0 is obtained by replacing those occurrences by x_1, ..., x_n,correspondingly. We shall develop an algorithm for transforming any expressions into expressions of the fully lazy normal form that satisfy the context condition.

It should be noted that expressions composed of combinators or super-combinators are of the fully lazy normal form. The combinator expression is simply the one that does not contain functions with local definitions.

3.2 Rewriting Where-Clauses

The first stage of lambda-hoisting is to transform **where**-clauses in the source expression into **whererec**-clauses by renaming variables according to the rules shown in Fig. 3. By doing this, we become free from worries concerning the conflict of identifiers that may be caused when free occurrences of expressions are moved to outside the function from where they originally appear.

Although we do not give a formal definition of *fresh variables* for brevity, the next proposition should be observed.

Proposition
For any $\pi \in \mathbf{R}$, $x \neq y$ and a fresh variable $x' \in \mathbf{Ide}$, $(\pi + \langle x \to x' \rangle)[y] \neq x'$.

Using this, we can prove that

Lemma
For any $\pi \in \mathbf{R}$ and $\rho, \rho' \in \mathbf{U}$ satisfying $\rho'(\pi[x]) = \rho[x]$,

$$E[R[e]\pi]\rho' = E[e]\rho$$

holds for any $e \in \mathbf{Exp}$.

Since initial environments $\pi_0 \in \mathbf{R}$ and $\rho_0 \in \mathbf{U}$ satisfy $\rho_0(\pi_0[x]) = \rho_0[x]$, the next theorem holds.

Theorem
$E[R[e]\pi_0]\rho_0 = E[e]\rho_0$ for any program e.

The theorem states that the meaning of program is preserved through the transformation by the rules in Fig. 3.

Environment for renaming

$\pi \in \mathbf{R} = [\mathbf{Ide} \to \mathbf{Ide}_\perp]$

Rewriting rules

R: $\mathbf{Exp} \to \mathbf{R} \to \mathbf{Exp}$

$R[b]\pi = b$
$R[x]\pi = \pi[x]$
$R[e_0\, e_1]\pi = (R[e_0]\pi)(R[e_1]\pi)$
$R[\mathbf{fn}\ x.\ e_0]\pi = \mathbf{fn}\ x'.\ R[e_0](\pi + \langle x \to x' \rangle)$
where x' is a fresh identifier
$R[e_0\ \mathbf{where}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\pi =$
$R[e_0]\pi'\ \mathbf{whererec}\ x_1' = R[e_1]\pi\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n' = R[e_n]\pi$
where $\pi' = \pi + \langle x_1 \to x_1' \rangle + \ldots + \langle x_n \to x_n' \rangle$
and x_i' are fresh identifiers
$R[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\pi =$
$R[e_0]\pi'\ \mathbf{whererec}\ x_1' = R[e_1]\pi'\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n' = R[e_n]\pi'$
where $\pi' = \pi + \langle x_1 \to x_1' \rangle + \ldots + \langle x_n \to x_n' \rangle$
and x_i' are fresh identifiers

Notation

For an environment for renaming π, $\pi + \langle x \to x' \rangle$ denotes
$\lambda y.\mathbf{if}\ x = y\ \mathbf{then}\ x'\ \mathbf{else}\ \pi[y]$.

Initial Environment

The initial environment π_0 satisfies $\pi_0[x] = x$ for pre-defined identifiers x.

Fig. 3 Rules for renaming identifiers and rewriting **where**-clauses.

3.3 Computing Lexical Levels

The main part of lambda-hoisting is to identify maximal free occurrences of expressions (See Section 3.4). To do so, it is necessary to determine **fn**-variables on which each expression depends. Each **fn**-variable can be identified by the level number, i.e., the number of nested **fn**-abstractions. We assume that the outermost **fn**-variable is assigned the level number 1.

In Hughes' algorithm for finding super-combinators, each compound expression, or *combination,* of the form $(e_0\, e_1)$, is assigned the maximum level number of its constituents. It is insufficient for our purpose, however. As it will turn out, it is necessary to assign a set of level numbers to each expression. For a combination, the union of the sets of level numbers for its constituents is assigned. Every constant has the singleton set $\{0\}$, and each variable has $\{0\} \cup \{l\}$ where l is the level of the variable.

We denote the maximum of a set of level numbers

$$\overline{l} = \{l_1, l_2, \ldots, l_n\}$$

by

$$|\,\overline{l}\,| = \max\{l_1, l_2, \ldots, l_n\}$$

The rules for assigning the set of level numbers to the expression are shown in Fig. 4. Since lexical levels are computed for expressions tranformed by the rules

Level numbers

$l \in \mathbf{N}$

Environment for level numbers

$\omega \in \mathbf{L} = [\mathbf{Ide} \to \mathbf{N}_+]$

Assignment rules

L: $\mathbf{Exp} \to \mathbf{L} \to \mathbf{N} \to \mathbf{2^N}$

$L[b]\omega l = \{0\}$
$L[x]\omega l = \{0\} \cup \{\omega[x]\}$
$L[e_0\, e_1]\omega l = L[e_0]\omega l \cup L[e_1]\omega l$
$L[\mathbf{fn}\ x.\ e_0]\omega l = L[e_0](\omega + \langle x \to l+1\rangle)(l+1) - \{l+1\}$
$L[e_0\ \mathbf{where}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\omega l = L[e_0]\omega' l$
where $\omega' = \omega + \langle x_1 \to l_1\rangle + \ldots + \langle x_n \to l_n\rangle$
where $l_i = |\,L[e_i]\omega l\,|$ for $i = 1, \ldots, n$.
$L[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\omega l = L[e_0]\omega' l$
where $\omega' = \omega + \langle x_1 \to l_1\rangle + \ldots + \langle x_n \to l_n\rangle$
where $l_i = |\,L[e_i]\omega' l\,|$ for $i = 1, \ldots, n$.

Notation

For an environment for assignment ω, $\omega + \langle x \to l\rangle$ denotes
$\lambda y.\mathbf{if}\ x = y\ \mathbf{then}\ l\ \mathbf{else}\ \omega[y]$.

Initial environment

The initial environment ω_0 satisfies $\omega_0[x] = 0$ for pre-defined identifiers x.

Fig. 4 Rules for assigning level numbers to expressions.

in Section 3.2, the rule for **where**-abstraction is not used in our lambda-hoisting algorithm, though it is included in Fig. 4.

The rule for **fn**-abstractions is worth noting. Assume that **fn** x: e_0 appears in the context $\omega \in \mathbf{L}$ of the level l. If e_0 contains x and no other variables, we get

$$|L[\mathbf{fn}\ x\text{: } e_0]\omega l|=0$$

from the rule. Hence the combinator has the level 0 in any context and at any level. For example, an occurrence of a combinator

$$\mathbf{S}=\mathbf{fn}\ f\ g\ x\text{: } f\ x(g\ x)$$

is considered as a constant.*

Consider another case that e_0 contains a variable y of $\omega[y]=l$ as in

$$\mathbf{fn}\ y\text{: }\{\mathbf{fn}\ x\text{: }\{\ldots\ y\ \ldots\}\}$$

We have

$$L[e_0](\omega+\langle x \rightarrow l+1\rangle)(l+1)=\{0,\ l,\ \ldots\}$$

and it is concluded that

$$|L[\mathbf{fn}\ x\text{: } e_0]\omega l|=l$$

whether x appears in e_0 or not. That is, the function **fn** x: e_0 depends on the variable of level l. This example illustrates the reason why we compute the set of lexical levels instead of only the maximal level. It is necessary to find the largest or the second largest of the level numbers assigned to e_0.

Finally, it should be noted that the rule for **whererec**-abstractions is stated using a recursive equation for ω'.

$$\omega'=\omega+\langle x_1 \rightarrow |L[e_1]\omega' l|\rangle+\ \ldots\ +\langle x_n \rightarrow |L[e_n]\omega' l|\rangle$$

A question may arise: does the equation have a solution at all? If it does, we need an algorithm to find an ω' satisfying the equation. We answer this by presenting a simple algorithm. This can be used in practice for lambda-hoisting, though it is not optimal. The algorithm is based on a simple iteration starting with initial approximations to $|L[e_i]\omega' l|$ and improving them successively.

l_i':=0 for $i=1, \ldots, n$;
repeat
    l_i:=l_i' for $i=1, \ldots, n$;
    ω':=$\omega+\langle x_1 \rightarrow l_1\rangle+\ \ldots\ +\langle x_n \rightarrow l_n\rangle$;
    l_i':=$|L[e_i]\omega' l|$ for $i=1, \ldots, n$;
until $\{l_i=l_i'$ for every $i=1, \ldots, n\}$

* In the extended **fn**-binding **fn** $x_1 \ldots x_n$, we consider that variables $x_1, \ldots, x_n$ are of the same level, say $(l+1)$ if the **fn**-abstraction appears at level l.

From the observation that the operations employed are monotonic, and $0 \leq l_i \leq l$ holds for $i=1, \ldots, n$, it can be shown that the algorithm terminates.

3.4 Hoisting Maximal Free Occurrences of Combinations

We now define *free occurrences* of combinations. The free occurrence of combinations is a simple extension of the free occurrence of variables that is not bound by an **fn**-abstraction. Note that we deal with expressions transformed by the rules in Section 3.2.

Definition (Free occurrences of combinations)
An occurrence of an expression of the form $(e_0\ e_1)$ is called a *free occurrence* with respect to $\omega \in \mathbf{L}$ and $l \in \mathbf{N}$, if

$$0 \leq |L[e_0]\omega l| < l, \quad 0 \leq |L[e_1]\omega l| < l, \quad \text{and } |L[e_0 e_1]\omega l| \neq 0$$

hold.

Since

$$|L[e_0 e_1]\omega l| = |L[e_0]\omega l \cup L[e_1]\omega l| = |L[e_0]\omega l| \text{ or } |L[e_1]\omega l|$$

from the rule in Fig. 4, the above condition can be restated as: both $|L[e_0]\omega l|$ and $|L[e_1]\omega l|$ are less than l, but at least either of them is greater than 0.

Definition (Maximal free occurrences of combinations)
A free occurrence of a combination $e^* = (e_0\ e_1)$ with respect to $\omega \in \mathbf{L}$ and $l \in \mathbf{N}$ is called *maximal*, if either of following conditions holds.

(1) There is an occurrence of a combination containing e^* as $(e' e^*)$ or $(e^*\ e')$, and

$$|L[e^*]\omega l| < |L[e']\omega l|$$

holds.

(2) The occurrence e^* appears as either

fn x: e^*,
e^* **whererec** $x_1 = e_1$ **and** ... **and** $x_n = e_n$, or

$x_i = e^*$ in a **whererec**-clause.

Rules for lambda-hoisting are shown in Fig. 5. In short, maximal free occurences of combinations are moved outside the original **fn**-body by creating new declarations with fresh variables.

Algorithm
An expression e is transformed into e^* of the fully lazy normal form by

$$\langle \mu^*, \omega^*, e^* \rangle = H[R[e]\pi_0]\mu_0 0$$

Level numbers

$l \in \mathbf{N}$

Environment for level numbers

$\omega \in \mathbf{L}$

Declarations of maximal free occurrences of combinations

$\mu \in \mathbf{M} = [\mathbf{N} \to 2^{\mathbf{Dec}}]$

$d = \mathbf{Dec}$ declarations

$d ::= x = e$

Hoisting rules

H: $\mathbf{Exp} \to \mathbf{M} \to \mathbf{L} \to \mathbf{N} \to [\mathbf{M} \times \mathbf{L} \times \mathbf{Exp}]$

$H\,[b]\,\mu\omega l = \langle \mu,\ \omega,\ \lceil b \rceil \rangle$

$H\,[x]\,\mu\omega l = \langle \mu,\ \omega,\ \lceil x \rceil \rangle$

$H\,[e_0\ e_1]\,\mu\omega l = \langle \mu^*,\ \omega^*,\ e^* \rangle$

let $\langle \mu'',\ \omega'',\ e_1' \rangle = H\,[e_1]\,\mu'\omega' l$ where $\langle \mu',\ \omega',\ e_0' \rangle = H\,[e_0]\,\mu\omega l$ in

if $e_i'\ (i = 0, 1)$ is a maximal free occurrence of a combination w.r.t. ω'' and l,

$\mu^* = \mu'' + \langle k \to \mu'' k \cup \lceil x' = e_i' \rceil \rangle$, $\omega^* = \omega'' + \langle x' \to k \rangle$,

and $e^* = \lceil (x'\,e_1') \rceil$ or $e^* = \lceil (e_0'\,x') \rceil$ for $i = 0, 1$, respectively,

where $k = |L\,[e_i]\,\omega'' l|$ and x' is a fresh identifier

else $\mu^* = \mu''$, $\omega^* = \omega''$ and $e^* = (e_0'\,e_1')$

$H\,[\mathbf{fn}\ x.\ e_0]\,\mu\omega l = \langle \mu^*,\ \omega^*, \lceil \mathbf{fn}\ x.\ e_0^* \rceil \rangle$

let $\langle \mu',\omega',\ e_0' \rangle = H\,[e_0]\,(\mu + \langle l+1 \to \{\ \} \rangle)(\omega + \langle x \to l+1 \rangle)(l+1)$ in

if $\mu'(l+1) = \{\ \}$

and e_0' is a maximal free occurrence of a combination w.r.t. ω' and l,

$\mu^* = \mu' + \langle k \to \mu' k \cup \lceil x' = e_0' \rceil \rangle$, $\omega^* = \omega' + \langle x' \to k \rangle$, and $e^* = \lceil x' \rceil$

where $k = |L\,[e_0']\,\omega' l|$ and x' is a fresh indentifier

else $\mu^* = \mu'$, $\omega^* = \omega'$, and $e^* = \lceil e_0'\ \mathbf{whererec}\ \mu'(l+1) \rceil$

$H\,[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\,\mu\omega l = H\,[e_0]\,\mu_n\omega_n l$

where $\mu_i = \mu_i' + \langle k \to \mu_i' \cup \lfloor x_i = e_i' \rfloor \rangle$ and $\omega_i = \omega_i' + \langle x_i \to k \rangle$

where $\langle \mu_i',\ \omega_i'\ e_i' \rangle = H\,[e_i]\,\mu_{i-1}\omega_{i-1} l$ and $k = |L\,[e_i']\,\omega_i' l|$

for $i = 1, \ldots, n$, and $\mu_0 = \mu$, $\omega_0 = \omega$

Notations

Tuples in $[\mathbf{M} = \mathbf{L} \times \mathbf{Exp}]$ are written as $\langle \mu,\ \omega,\ e \rangle$.

Syntactic elements are quoted by $\lceil$ and $\rceil$.

For a declaration set μ, $\mu + \langle k \to \sigma \rangle$ denotes

$\lambda l.\mathbf{if}\ j = k\ \mathbf{then}\ \sigma\ \mathbf{else}\ \mu j$

If $\mu l = \{\lceil x_1 = e_1 \rceil, \ldots, \lceil x_n = e_n \rceil\}$, $\lceil e_0\ \mathbf{whererec}\ \mu l \rceil$ denotes

$\lceil e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n \rceil$

Initial set of declarations

The initial set of declarations μ_0 satisfies $\mu_0 l = \{\ \}$ for any $l \in \mathbf{N}$.

Fig. 5 Lambda-hoisting rules for expressions of the fully lazy form.

```
Source program:
    fn x:
        {ξ FORK(fn u: TIP(ξ MIN 1))
            where ξ = btree x
                whererec
                    btree = fn x:
                            {fn g f:
                              IF(ISTIP x)(f(TIPVAL x))
                                (g(btree(LEFT x)) g f)
                                    (btree(RIGHT x) g f))}}

Fully lazy normal form:
    {fn x': {ξ' FORK(fn u': α)} whererec ξ' = btree' x' and α = TIP(ξ' MIN 1)}
    whererec
        btree' = fn x'': {fn g' f': β2(f' β1)(g'(β3 g' f')(β4 g' f'))}
                        whererec β1 = TIPVAL x'' and β2 = IF(ISTIP x'')
                          and β3 = btree'(LEFT x'')
                          and β4 = btree'(RIGHT x'')
```

Fig. 6 An example of lambda-hoisting.

For example,

$$\textbf{fn}\ x\text{: }\{\textbf{fn}\ y\text{: }(+(-\ x\ 1)\ y)\}$$

is transformed into*

$$\textbf{fn}\ x\text{: }\{\textbf{fn}\ y\text{: }(z\ y)\}\ \textbf{whererec}\ z=(+(-\ x\ 1))$$

A more realistic example is shown in Fig. 6. This example demonstrates the use of full laziness for eliminating multiple traversals of data structures.[14]

As the function *btree* in the source program has no free variables, it has been moved to the outermost level in the fully lazy form. The reader will have observed from this example how the lambda-hoisting algorithm works.

We beleive that the meaning of programs remains unchanged through the transformation. That is,

$$E[e^*]\rho_0 = E[e]\rho_0 \text{ where } \langle \mu^*, \omega^*, e^* \rangle = H[R[e]\pi_0]\mu_0 0$$

Although we have not completed the proof of this *soundness* theorem, it seems possible to prove it with great care in dealing with *strict* functions such as arithmetic operations and predicates. Then the lambda-hoisting transformation preserves the meaning of programs. Evaluation of the resulting program in a lazy way turns out to be fully lazy. Since full laziness implies ordinary laziness by definition, it would be obvious at least intuitively that the transformed program requires no more evaluation steps than the original. A careful study should be made on this point, however. It is not the scope of this paper.

* The form **fn** *x*:*e* **whererec** ... should be read as **fn** *x*: {*e* **whererec** ...}.

§4 Conclusion

In this paper we have developed an algorithm for lambda-hoisting. The algorithm presented in the previous section can be considered as a functional program, though there remain some informal descriptions like "... for $i=1, ..., n$". We have a compiler that translates programs written in an experimental functional language *uc* into fully lazy normal form. Programs of the fully lazy normal form are compiled into fixed code of the *Fully Lazy Functional Machine*.[13] The FLFM code is then translated into machine code for conventional computers. We have developed code generators for MC68000, i8086, Melcom 70/250, Melcom MX2000, and Fujitsu M-series computers. As the task of the code generator is simple macro processing driven by a table, it is easy to generate code for other machines.

The fully lazy normal form relaxes restrictions on the representation of functions; the function body possibly contains local definitions. The supercombinator approach keeps the traditional form based on lambda calculus, and **where**- and **whererec**- clauses are transformed into functional applications, e.g., e_0 **where** $x_1 = e_1$ into (**fn** x_1: e_0)e_1. Although these are semantically equivalent in our language as well, we have not followed this transformation because the number of parameter passing becomes large for nested function definitions.

The key to the lambda-hoisting technique is to transform programs into ones with *local recursion*. Elimination of non-recursive local declarations by **where**-clauses greatly simplifies the algorithm. A similar compilation technique called *lambda-lifting*[7,8] takes no account of full laziness. Functions expressed by **fn**-abstractions inside another function remain as they are by lambda-hoisting, while all the functions are made global by lambda-lifting. The presence of local functions enables one to instantiate a function to obtain other functions by partial parametrization.

We have presented an example that demonstrates the use of full laziness for eliminating multiple traversals of data structures. Fully lazy evaluation brings unexpected gains in efficiency. More investigation on the novel feature with relation to partial parametrization in functional programming is expected. The efficiencies of full laziness should be studied further.

Acknowledgements

I sincerely thank Eiiti Wada for his comments and suggestions on the first draft of the manuscript.

References

1) Friedman, D. P. and Wise, D. S., "CONS should not evaluate its arguments," *Proc. 3rd International Colloquium on Automata, Languages and Programming,* pp. 257-284, 1976.
2) Henderson, P. and Morris, J. M., "A lazy evaluator," *Proc. 3rd Symp. on Principle of Programming Languages,* pp. 95-103, 1976.
3) Henderson, P., *Functional Programming: Application and Implementation,* Prentice-Hall, 1980.
4) Henderson, P., Jones, G. A. and Jones, S. B., "The Lispkit Manual," *Technical Monograph PRG-32,* Oxford University Computing Laboratory, 1983.
5) 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.
6) Hughes, R. J. M., "The desing and implementation of programming languages," *D. Phil. thesis,* Oxford University, 1984.
7) Johnsson, T., "Efficient compilation of lazy evaluation," *Proc. SIGPLAN' 84 Symp. on Compiler Construction,* pp. 58-69, 1984.
8) Johnsson, T., "Lambda-lifting: Transforming Programs to Recursive Equations," *Lecture Notes in Computer Science 201,* Springer-Verlag, pp. 190-203, 1985.
9) 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.
10) Landin, P. J., "The next 700 programming languages," *Comm. ACM,* pp. 157-164, 1966.
11) Takeichi, M., "Evaluation of combinator expressions," *Proc. 1st JSSST Annual Conference* [in Japanese], pp. 213-222, 1984.
12) Takeichi, M., "An Alternative Scheme for Evaluating Combinator Expressions", *Journal of Information Processing,* pp. 246-253, 1985.
13) Takeichi, M., "A Functional Machine for Fully Lazy Evaluation (Extended Abstract)," *Proc. RIKEN Symp. on Functional Programming,* pp. 54-60, 1986.
14) Takeichi, M., "Partial Parametrization Eleminates Multiple Traversals of Date Structures," *Acta Informatica, 24,* pp. 57-77, 1987.
15) Turner, D. A., "SASL language manual", *St. Andrew's University Technical Report No. CS/75/1,* 1976.
16) Turner, D. A., "A New Implementation Technique for Applicative Languages," *Software—Practice and Experience,* pp. 39-49, 1979.
17) Turner, D. A., "Aspects of the implementation of programming languages," *D. Phil. thesis,* Oxford University, 1981.
18) Turner, D. A., "Recursion equations as a programming language," in *Functional Programming and Its Applications* (J. Darlington, P. Henderson and D. A. Turner, eds.), Cambridge University Press, pp. 1-28, 1982.
19) Turner, D. A., "Miranda: A non-strict functional language with polymorphic types", *Lecture Notes in Computer Science, 201,* Springer-Verlag, pp. 1-16, 1985.
20) Vuillemin, J., "Correct and optimal implementations of recursion in a simple programming language," *J. Comp. Sys. Sci., 9,* pp. 332-354, 1974.
21) Wadsworth, C. P., "Semantics and Pragmatics of the Lambda-Calculus," *D. Phil. thesis,* Oxford University, 1971.