Regular Paper

Relationship between Lambda Hoisting and Fully Lazy Lambda Lifting

KEIICHI KANEKO* and MASATO TAKEICHI*

Two algorithms have been proposed for transforming functional programs into ones suitable for fully lazy evaluation—Lambda hoisting (Takeichi [7]) and fully lazy lambda lifting (Peyton Jones [4]). These algorithms share similar operations such as floating out local definitions and extracting maximal free occurrences of subexpressions to achieve full laziness, while they look different at first sight. This paper investigates these algorithms in a same framework and shows that the differences lie in the evaluation schemes for the resultant programs. We conclude that the main part of these algorithms may be considered identical by transforming the lambda hoisting rules into those for fully lazy lambda lifting.

1. Introduction

First consider an **fn**-expression:[1]

$$\textbf{fn}\ x\text{: I (}\textbf{fn}\ y\text{: } y\ a\ (+\ x\ 1)\ \textbf{whererec}\ a=(+\ x\ 2))$$
$$(\mathrm{K}\ (\textbf{fn}\ z\text{: } *\ z\ b\ \textbf{whererec}\ b=(+\ x\ 3)))$$

where I and K are combinators. This expression is transformed using fully lazy lambda lifting into a supercombinator Ψ defined as:

$$\Psi\ x=\mathrm{I}\ (\Phi_1\ a(+\ x\ 1)\ \textbf{whererec}\ a=+\ x\ 2)$$
$$(\mathrm{K}\ (\Phi_2\ b\ \textbf{whererec}\ b=+\ x\ 3)) \qquad (1)$$

with auxiliary supercombinators:

$$\Phi_1\ p\ q\ y=y\ p\ q \text{ and}$$
$$\Phi_2\ r\ z=*\ z\ r.$$

On the other hand, by lambda hoisting, we have:

$$\textbf{fn}\ x\text{: I (}\textbf{fn}\ y\text{: } (y\ a\ y))(\mathrm{K}\ (\textbf{fn}\ z\text{: } (*\ z\ b)))$$
$$\textbf{whererec}\ a=(+\ x\ 2)$$
$$\textbf{and}\ b=(+\ x\ 3)\ \textbf{and}\ y=(+\ x\ 1). \qquad (2)$$

Comparing (1) and (2), we observe the expressions, Ψ, $\Phi_1\ a(+\ x\ 1)$, and $\Phi_2\ b$, correspond to **fn**-expressions, **fn** x: I(. . .)(. . .), **fn** y: $(y\ a\ y)$, and **fn** z: $(*\ z\ b)$, respectively. The resultant expression (2) differs from (1) in two points:

- Local definitions are collected into a single **whererec**-clause.
- Each maximal free occurrence of a subexpression is treated as a local definition.

In Section 2, a simple functional language is introduced to describe the algorithms. Then we investigate the reason why these differences appear in Section 3. Finally we transform the lambda hoisting rules into those for fully lazy lambda lifting in Section 4.

2. Preliminaries

2.1 A Simple Functional Language

We introduce a simple functional language to provide a common basis for describing the algorithms. Figure 1 shows a denotational specification of our language. The reader may wonder why **where**-clauses are missing in it, while **whererec**-clauses are included. In fact, we can implement any local definitions with **whererec**, but we cannot with **where**. We also assume that **fn**-variables and locally defined variables are all distinct, and no names may crash in the course of transformation. In summary, functional programs written in a language with more general features are supposed to be transformed into ones in our language before they are converted into fully lazy ones.

2.2 Lambda Hoisting

The lambda hoisting algorithm attains full laziness by transforming an expression in our functional language into one of more restricted form called the fully lazy normal form shown in Fig. 2. In the direct consequence of the context condition, lazy evaluation of arguments and local definitions brings full laziness.

In order to hoist free occurrences of expressions, their lexical levels should be calculated. We can deter-

*Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.

[1] We use **fn** instead of λ, because we use λ in semantic explanation.

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: $e \mid e$ **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
$\mathbf{U}=\mathbf{Ide}\to\mathbf{D}_{\perp}$ environment
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: $e_0]\rho=\lambda\delta.\ E[e_0](\rho+\langle x\to\delta\rangle)$
$E[e_0$ **whererec** $x_1=e_1$ **and** . . . **and** $x_n=e_n]\rho=E[e_0]\rho'$
where $\rho'=\rho+\langle x_1\to E[e_1]\rho'\rangle+\ \ldots\ +\langle x_n\to E[e_n]\rho'\rangle$
Notations
Operator + stands for the disjoint sum.
For any domain **X**, $\mathbf{X}_{\perp}=\mathbf{X}+\{$**err**$\}$.
For an environment ρ, $\rho+\langle x\to\delta\rangle$ denotes
$\lambda y.$ **if** $x=y$ **then** δ **else** $\rho[y]$.
Initial Environment
For pre-defined identifiers x, ρ_{ϕ} satisfies $\rho_{\phi}[x]\neq$**err**.

Fig. 1 Specification of Our Functional Language.

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
e contains no free occurrence of compound expressions.

Fig. 2 Fully Lazy Normal Form.

mine whether each subexpression is free or bound in the expression from its lexical level. Each variable is assigned a level number which corresponds to the depth of nested **fn**-abstractions. By definition, every basic value has level number zero. The level number of an expression is used to find variables on which each subexpression depends.

In Hughes [2] approach, only the maximum level number of the constituent expressions is used for finding maximal free occurrences of expressions. In lambda hoisting, we assign every subexpression with a set of level numbers of its constituents to determine how many levels the maximal free occurrences of subexpressions are hoisted to .

The rule for **whererec**-clauses is represented by a recursive equation. There is a not optimal but simple algorithm to solve it. See Takeichi [5] for details.

We need a definition for maximal free occurrences of combinations.

Definition (Maximum of a Set of Level Numbers)
For any set of level numbers $\bar{l}=\{l_1, l_2, \ldots, l_n\}$, maximum of the set is denoted by $|\bar{l}|=\max\{l_1, l_2, \ldots, l_n\}$. ■

Definition (Free Occurrences of Combinations)

Level Numbers
$l \in \mathbf{N}$
Environment for Level Numbers
$\omega \in \mathbf{L}=[\mathbf{Ide}\to\mathbf{N}_{+}]$
Assignment Rules
L: **Exp**$\to$**L**$\to$**N**$\to 2^{\mathbf{N}}$
$L[b]\omega l=\{0\}$
$L[x]\omega l=\{0\}\cup\{\omega[x]\}$
$L[e_0 e_1]\omega l=L[e_0]\omega l\cup L[e_1]\omega l$
$L[$**fn** x: $e_0]\omega l=L[e_0](\omega+\langle x\to l+1\rangle)(l+1)-\{l+1\}$
$L[e_0$ **whererec** $x_1=e_1$ **and** . . . **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 ω, $\omega+\langle x\to l\rangle$ denotes
$\lambda y.$ **if** $x=y$ **then** l **else** $\omega[y]$.
Initial Environment
For pre-defined identifiers x, ω_{ϕ} satisfies $\omega_{\phi}[x]=0$.

Fig. 3 Rules for Assigning Level Numbers.

Declarations of Maximal Free Occurrences of Combinations
$\mu \in \mathbf{M}=[\mathbf{N}\to 2^{\mathbf{Dec}}]$
$d \in$ **Dec** declarations
$d ::= x=e$
Hoisting Rules
H: **Exp**$\to$**M**$\to$**L**$\to$**N**$\to[$**M** $\times$ **L** $\times$ **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'(either $i=0$ or 1) is an MFOC w.r.t. ω'' and $l+1$,
$\mu^*=\mu''+\langle k\to\mu'' k\cup\lceil x'=e_i'\rceil\rangle$, $\omega^*=\omega''+\langle x'\to k\rangle$,
and $e^*=\lceil(x' e_i')\rceil$ 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 $e^*=(e_0' e_1')$
$H[$**fn** x: $e_0]\mu\omega l=\langle\mu^*, \omega^*, \lceil$**fn** x: $e^*\rceil\rangle$
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 an MFOC 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 identifier
else $\mu^*=\mu'$, $\omega^*=\omega'$, and $e^*=\lceil e_0'$ **whererec** $\mu'(l+1)\rceil$
$H[e_0$ **whererec** $x_1=e_1$ **and** . . . **and** $x_n=e_n]\mu\omega l=H[e_0]\mu_n\omega_n l$
where $\mu_i=\mu_i'+\langle k_i\to\mu'\cup\lceil x_i=e_i'\rceil\rangle$ and $\omega_i=\omega_i'+\langle x_i\to k_i\rangle$
where $\langle\mu_i', \omega_i', e_i'\rangle=H[e_i]\mu_{i-1}\omega_{i-1} l$ and $k_i=|L[e_i']\ \omega_i' l|$
for $i=1, \ldots, n$, and $\mu_0=\mu$, $\omega_0=\omega$
Notations
Tuples in **[M** $\times$ **L** $\times$ **Exp]** are written as $\langle\mu, \omega, e\rangle$.
Syntactic elements are quoted by $\lceil$ and $\rceil$.
For a declaration set μ, $\mu+\langle k\to v\rangle$ denotes
$\lambda j.$ **if** $j=k$ **then** v **else** μj
If $\mu l=\{\lceil x_1=e_1\rceil, \ldots, \lceil x_n=e_n\rceil\}$, $\lceil e_0$ **whererec** $\mu l\rceil$ denotes
$\lceil e_0$ **whererec** $x_1=e_1$ **and** . . . **and** $x_n=e_n\rceil$
Initial Set of Declarations
The initial set of declarations μ_{ϕ} satisfies $\mu_{\phi} l=\{\}$ for any $l \in \mathbf{N}$.

Fig. 4 Lambda Hoisting Rules.

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 $0 \le |L[e_0]\omega l| < l$, $0 \le |L[e_1]\omega l| < l$, and $|L[e_0 e_1]\omega l| \neq 0$ hold. ■

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 the 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. ■

Now an expression e is transformed into (e^* **whererec** $\mu^* 0$) in the fully lazy normal form by the lambda hoisting rules:

$$\langle \mu^*, \omega^*, e^* \rangle = H[e]\mu_\phi \omega_\phi 0.$$

2.3 Fully Lazy Lambda Lifting

We follow the description of fully lazy lambda lifting in Peyton Jones [4]. However, the algorithm is simplified for brevity in this paper.

We show how an expression in our functional language is transformed by fully lazy lambda lifting into declarations of supercombinators. Full laziness is achieved by floating local definitions outwards and by abstracting maximal free occurrences of expressions using supercombinators. Thus the algorithm breaks into two phases. It floats out the local definitions as far as possible in the first phase, it detects and abstracts the maximal free occurrences of expressions in the second phase. This approach differs from Peyton Jones [6] where these operations are performed in reverse order.

The definitions for maximal free occurrences of expressions and supercombinators follow:

Definition (Free Occurrences of Expressions)
An expression e is called free with respect to $\omega \in \mathbf{L}$ and $l \in \mathbf{N}$, if $0 \leq |L[e]\omega l| < l$ holds. ■

Definition (Maximal Free Occurrences of Expressions)
A free occurrence of an expression e^* with respect to $\omega \in \mathbf{L}$ and $l \in \mathbf{N}$ is called maximal, if either of the following conditions holds.

1) There is an occurrence of a combination containing e^* as $(e'\ e^*)$ or $(e^*\ e')$, and $|L[e']\omega l| = l$ holds.

2) The expression 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. ■

Definition (Supercombinators)
An expression e which has no free variable is called a supercombinator, if e is in the form of **fn** x_1: **fn** x_2: . . . **fn** x_n: e', and e' does not contain any lambda abstraction which is not a supercombinator. ■

Floating Rules

F: **Exp**→**M**→**L**→**N**→[**M** × **L** × **Exp**]
$F[b]\mu\omega l = \langle \mu, \omega, \lceil b \rceil \rangle$
$F[x]\mu\omega l = \langle \mu, \omega, \lceil x \rceil \rangle$
$F[e_0 e_1]\mu\omega l = \langle \mu^*, \omega^*, e^* \rangle$
  let $\langle \mu', \omega', e_0' \rangle = F[e_0]\mu\omega l$ in
    $e^* = \lceil (e_0' e_1') \rceil$
      where $\langle \mu^*, \omega^*, e_1' \rangle = F[e_1]\mu'\omega' l$
$F[$**fn** x: $e_0]\mu\omega l = \langle \mu^*, \omega^*, \lceil$**fn** x: $e_0^* \rceil \rangle$
  where $e_0^* = \lceil e_0'$ **whererec** $\mu^*(l+1) \rceil$
    where $\langle \mu^*, \omega^*, e_0' \rangle$
      $= F[e_0](\mu + \langle l+1 \to \{\} \rangle)(\omega + \langle x \to l+1 \rangle)(l+1)$
$F[e_0$ **whererec** $x_1 = e_1$ **and** . . . **and** $x_n = e_n]\mu\omega l = F[e_0]\mu_n\omega_n l$
  where $\mu_i = \mu_i' + \langle k \to \mu' k \cup \{ \lceil x_i = e_i' \rceil \} \rangle$ and $\omega_i = \omega_i' + \langle x_i \to k \rangle$
    where $\langle \mu_i', \omega_i', e_i' \rangle = F[e_i]\mu_{i-1}\omega_{i-1} l$ and $k = |L[e_i']\omega_i' l|$
  for $i = 1, \ldots, n$, and $\mu_0 = \mu$, $\omega_0 = \omega$

Fig. 5 Floating Rules for Local Definitions.

Abstracting Rules

A: **Exp**→**M**→**L**→**N**→[**M** × **L** × **Exp**]
$A[b]\mu\omega l = \langle \mu, \omega, \lceil b \rceil \rangle$
$A[x]\mu\omega l = \langle \mu, \omega, \lceil x \rceil \rangle$
$A[e_0 e_1]\mu\omega l = \langle \mu^*, \omega^*, e^* \rangle$
  let $\langle \mu', \omega', e_0' \rangle = A[e_0]\mu\omega l$ in
    let $\langle \mu'', \omega'', e_1' \rangle = A[e_1]\mu'\omega' l$ in
    if e_i' (either $i=0$ or 1) is an MFOC w.r.t. ω'' and $l+1$,
      $\mu^* = \mu'' + \langle k \to \mu'' k \cup \lceil x' = e_i' \rceil \rangle$, $\omega^* = \omega'' + \langle x' \to k \rangle$,
      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 $e^* = \lceil (e_0' e_1') \rceil$
$A[$**fn** x: $e_0]\mu\omega l = \langle \mu^*, \omega^*, \lceil$**fn** x: $e^* \rceil \rangle$
  let $\langle \mu', \omega', e_0' \rangle = A[e_0](\mu + \langle l+1 \to \{\} \rangle)(\omega + \langle x \to l+1 \rangle)(l+1)$ in
    if $\mu'(l+1) = \{\}$ and e_0' is an MFOC 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 identifier
    else $\mu^* = \mu'$, $\omega^* = \omega'$, and $e^* = \lceil e_0'$ **whererec** $\mu'(l+1) \rceil$
$A[e_0$ **whererec** $x_1 = e_1$ **and** . . . **and** $x_n = e_n]\mu\omega l = \langle \mu^*, \omega^*, e_0^* \rangle$
  where $e_0^* = [e_0'$ **whererec** $x_1 = e_1'$ **and** . . . **and** $x_n = e_n'$ **and** $\mu^* l \rceil$
    where $\langle \mu^*, \omega^*, e_0' \rangle = A[e_0]\mu_n'\omega_n' l$
      where $\langle \mu_i', \omega_i', e_i' \rangle = A[e_i]\mu_{i-1}'\omega_{i-1}' l$
        for $i = 1, \ldots, n$ and $\mu_0' = \mu$, $\omega_0' = \omega$

Fig. 6 Abstracting Rules for MFOCs.

The algorithm for floating out local definitions is as follows:

1) For each local definition, we compute the level number of the defined variable by computing its definition body. This level number identifies the innermost **fn**-abstraction on which the definition depends.
2) The definition then be floated out until the nearest enclosing **fn**-abstraction of which level is the level number of the definition.
3) If the definition appears in the function position of an application, it is floated out until it does not.

Identifying maximal free occurrences of expressions is performed in a single tree walk over the expression:

1) On the way down the tree, the level number of

Floating Rules

F': **Exp**→**L**→**N**→[**M**×**L**×**Exp**]

$F'[b]\omega l=\langle \mu_\phi, \omega, \lceil b \rceil \rangle$

$F'[x]\omega l=\langle \mu_\phi, \omega, \lceil x \rceil \rangle$

$F'[e_0e_1]\omega l=\langle \mu^*, \omega^*, e^* \rangle$

let $\langle \mu', \omega', e_0' \rangle = F'[e_0]\omega l$ in

$\mu^*=\mu'+(\mu''+\langle l \rightarrow \{\}\rangle)$ and $e^*=\lceil (e_0'(e_1' \textbf{ whererec } \mu' l)) \rceil$

where $\langle \mu'', \omega^*, e_1' \rangle = F'[e_1]\omega' l$

$F'[\textbf{fn } x\text{: } e_0]\omega l=\langle \mu^*, \omega^*, \lceil \textbf{fn } x\text{: } e_0^* \rceil \rangle$

where $e_0^*=\lceil e_0' \textbf{ whererec } \mu^*(l+1) \rceil$

where $\langle \mu^*, \omega^*, e_0' \rangle = F'[e_0](\omega+\langle x \rightarrow l+1\rangle)(l+1)$

$F'[e_0 \textbf{ whererec } x_1=e_1 \textbf{ and } \ldots \textbf{ and } x_n=e_n]\omega l=\langle \mu^*, \omega^*, e_0^* \rangle$

$\mu^*=\mu_0'+\sum_{i=1}^{n}(\mu_i'+\langle k_i \rightarrow \{x_i=\lceil e_i' \textbf{ whererec } \mu_i' k_i \rceil\}\rangle)$

where $\langle \mu_0', \omega^*, e_0^* \rangle = F'[e_0]\omega_n' l$

where $\langle \mu_i', \omega_i', e_i' \rangle = F'[e_i]\omega_{i-1}' k_i$

for $i=1, \ldots, n$ and $\omega_0'=\omega$, $k_i=|L[e_i]\omega l|$

Notation

For any environments μ_1 and μ_2, $\mu_1+\mu_2$ denotes $\lambda l.\ (\mu_1 l \cup \mu_2 l)$.

Fig. 7 Revised Floating Rules for Local Definitions.

Declaration of Supercombinators

$\sigma \in \mathbf{S}$

$s=\mathbf{S}$

$s::=\Phi x \ldots x=e$

Abstracting Rules

A': **Exp**→**M**→**L**→**N**→**S**→[**M**×**L**×**Exp**×**S**]

$A'[b]\mu\omega l\sigma=\langle \mu, \omega, \lceil b \rceil, \sigma \rangle$

$A'[x]\mu\omega l\sigma=\langle \mu, \omega, \lceil x \rceil, \sigma \rangle$

$A'[e_0e_1]\mu\omega l\sigma=\langle \mu^*, \omega^*, e^*, \sigma^* \rangle$

let $\langle \mu', \omega', e_0', \sigma' \rangle = A'[e_0]\mu\omega l\sigma$ in

if e_i' (either $i=0$ or 1) is an MFOE w.r.t. ω'' and l,

if there exists k such that $\lceil x'=e_i' \rceil \in \mu'' k$

$\mu^*=\mu''$, $\omega^*=\omega''$, and $e^*=\lceil (x'e_i') \rceil$ or $e^*=\lceil (e_0'x') \rceil$

for $i=0$, 1, respectively,

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

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 $e^*=\lceil (e_0'e_1') \rceil$

where $\langle \mu'', \omega'', e_1', \sigma^* \rangle = A'[e_1]\mu'\omega' l\sigma'$

$A'[\textbf{fn } x\text{: } e_0]\mu\omega l\sigma=A'[e_0^*]\mu\omega l\sigma^*$

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

if e_0' is an MFOE w.r.t. ω' and $l+1$,

$\omega^*=\omega'+\langle \Phi \rightarrow 0\rangle$, $e_0^*=\lceil \Phi e_0' \rceil$

and $\sigma^*=\sigma' \cup \{\lceil \Phi x' x=x' \rceil\}$

where Φ and x' are fresh identifiers

else $\omega^*=\omega'+\langle \Phi \rightarrow 0\rangle$, $e_0^*=\lceil \Phi e_{01}' \ldots e_{0n_0}' \ldots e_{l1}' \ldots e_{ln_l}' \rceil$

and $\sigma^*=\sigma' \cup \lceil \Phi x_{01}' \ldots x_{0n_l}' \ldots x_{0l}' \ldots x_{0n_l}' x=e_0' \rceil$

where Φ is a fresh identifier

and $\mu' k$ is $\{\lceil x_{k1}'=e_{k1}' \rceil, \ldots, \lceil x_{kn_k}'=e_{kn_k}' \rceil\}$

$A'[e_0 \textbf{ whererec } x_1=e_1 \textbf{ and } \ldots \textbf{ and } x_n=e_n]\mu\omega l\sigma=\langle \mu^*, \omega^*, e_0^*, \sigma \rangle$

where $e_0^*=\lceil e_0' \textbf{ whererec } x_1=e_1' \textbf{ and } \ldots \textbf{ and } x_n=e_n' \rceil$

where $\langle \mu^*, \omega^*, e_0', \sigma^* \rangle = A'[e_0]\mu_n'\omega_n' l\sigma_n'$

where $\langle \mu_i', \omega_i', e_i', \sigma_i' \rangle = A'[e_i]\mu_{i-1}'\omega_{i-1}' l\sigma_{i-1}'$

for $i=1, \ldots, n$ and $\mu_0'=\mu$, $\omega_0'=\omega$, $\sigma_0'=\sigma$

Fig. 8 Revised Abstracting Rules for MFOEs.

each **fn**-variable is recorded.

2) On the way up, the level of each expression is computed, using the environment and the level of its subexpressions. If an expression turns out to be a maximal free occurrence of an expression, it is given a new fresh identifier.
3) When an **fn**-abstraction is encountered on the way up, it is transformed into a supercombinator, and the **fn**-abstraction is replaced by the supercombinator applied to maximal free occurrences of expressions in it.

3. Differences of the Algorithms

3.1 Local Definitions

Both algorithms allow local definitions in the source language. The algorithms float out the definitions as high as possible subject to the binding scope rule to attain full laziness. One of the difference is that after floating out the local definitions, lambda hoisting collects local definitions of the same level, while fully lazy lambda lifting leaves them separated. This difference originates from the difference of implementation schemes. That is, lambda hoisting adopts the environment model such as the SECD machine for evaluation. For example, we extract the local definitions in an expression

$$(\ldots((\ldots x \ldots) \textbf{ whererec } x=E) \ldots y \ldots) \textbf{ whererec } y=F,$$

to get the hoisted expression

$$(\ldots(\ldots x \ldots) \ldots y \ldots) \textbf{ whererec } x=E \textbf{ and } y=F.$$

The environment is updated only once when the expression is evaluated. In an actual implementation, each local definition is represented by a closure which is a pair consisting of a pointer to the code for the definition body and a pointer to an environment under which the code is executed. So unnecessary memory consumption is very little even if the variables x and y are not used in evaluation. In lambda hoisting, we should collect the local definitions so that we can avoid frequent update of the environment.

Fully lazy lambda lifting adopts graph reduction model based on recursive supercombinators. If we collect local definitions in an expression following the rules of lambda hoisting

$$\text{IF } B \ ((\ldots x \ldots) \textbf{ whererec } x=E) \ ((\ldots y \ldots) \textbf{ whererec } y=F),$$

we get

$$\text{IF } B \ (\ldots x \ldots)(\ldots y \ldots) \textbf{ whererec } x=E \textbf{ and } y=F.$$

When we evaluate this expression in graph reduction, we must always make two graphs for E and F regardless the evaluation result of the expression B. In fully lazy lambda lifting, therefore, we should float out local definitions no further than is necessary so that we can avoid constructing unnecessary graphs. Another version of fully lazy lambda lifter (Peyton Jones [6]) loses this property. In fact, the program in the version pro-

Lifting Rules

L': **Exp**→**M**→**L**→**N**→**S**→[**M** × **M** × **L** × **Exp** × **S**]

$L'[b]v\omega l\sigma = \langle \mu_\phi, \nu, \omega, \lceil b \rceil, \sigma \rangle$

$L'[x]v\omega l\sigma = \langle \mu_\phi, \nu, \omega, \lceil x \rceil, \sigma \rangle$

$L'[e_0 e_1]v\omega l\sigma = \langle \mu^*, \nu^*, \omega^*, e^*, \sigma^* \rangle$

let $\langle \mu', \nu', \omega', e_0', \sigma' \rangle = L'[e_0]v\omega l\sigma$ in

$\mu^* = \mu' + (\mu'' + \langle l \rightarrow \{\} \rangle)$,

if e_i' (either $i=0$ or 1) is an MFOE w.r.t. ω'' and l,

if there exists k such that $\lceil x' = e_i' \rceil \in \nu'' k$

$\nu^* = \nu''$, $\omega^* = \omega''$, and $e^* = \lceil (x'(e_1' \textbf{ whererec } \mu'' l)) \rceil$

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

else $\nu^* = \nu'' + \langle k \rightarrow \nu'' k \cup \{ \lceil x' = e_i' \rceil \} \rangle$, $\omega^* = \omega'' + \langle x' \rightarrow k \rangle$,

and $e^* = \lceil (x'(e_1' \textbf{ whererec } \mu'' l)) \rceil$ 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 $\nu^* = \nu''$, $\omega^* = \omega''$ and $e^* = \lceil (e_0'(e_1' \textbf{ whererec } \mu'' l)) \rceil$

where $\langle \mu'', \nu'', \omega'', e_1', \sigma^* \rangle = L'[e_1]\nu'\omega' l\sigma'$

$L'[\textbf{fn } x\text{: } e_0]v\omega l\sigma = \langle \mu^*, \nu^*, \omega^*, e_0^*, \sigma^* \rangle$

let $\langle \mu', \nu', \omega', e_0', \sigma' \rangle = L'[e_0]\nu_\phi(\omega + \langle x \rightarrow l+1 \rangle)(l+1)\sigma$ in

$\mu^* = \mu' + \mu''$

where $\langle \mu'', \nu^*, \omega^*, e_0^*, \sigma^* \rangle = L'[e_0'']\nu_\phi \omega'' l\sigma''$

where if e_0' is an MFOE w.r.t. ω' and $l+1$,

$\omega'' = \omega' + \langle \Phi \rightarrow 0 \rangle$, $e_0'' = \lceil (\Phi e_0') \rceil$,

and $\sigma'' = \sigma' \cup \{ \lceil \Phi x' x = x' \rceil \}$

else $\omega'' = \omega' + \langle \Phi \rightarrow 0 \rangle$,

$e_0'' = \lceil (\Phi e_{01}' \ldots e_{0n_0}' \ldots e_{l1}' \ldots e_{ln_l}') \rceil$

and $\sigma'' = \sigma' \cup \{ \lceil \Phi x_{01}' \ldots x_{0n_0}' \ldots x_{l1}' \ldots x_{ln_l}' = e_0' \textbf{ whererec } \mu'(l+1) \rceil \}$

where Φ is a fresh identifier

and νk is $\{ \lceil x_{k1}' = e_{k1}' \rceil \ldots \lceil x_{kn_k}' = e_{kn_k}' \rceil \}$

$L'[e_0 \textbf{ whererec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n]v\omega l\sigma = \langle \mu^*, \nu^*, \omega^*, e_0^*, \sigma \rangle$

where $\mu^* = \mu_0' + \sum_{i=1}^{n} (\mu_i' + \langle k_i \rightarrow \{ \lceil x_i = e_i' \textbf{ whererec } \mu_i' k \rceil \} \rangle)$

where $\langle \mu_0', \nu^*, \omega^*, e_0^*, \sigma^* \rangle = L'[e_0]\nu_n' \omega_n' l\sigma_n'$

where $\langle \mu_i', \nu_i', \omega_i', e_i', \sigma_i' \rangle = L'[e_i]\nu_{i-1}' \omega_{i-1}' k_i \sigma_{i-1}'$

for $i=1, \ldots, n$ and $\mu_0' = \mu$, $\omega_0' = \omega$, $\sigma_0' = \sigma$, $k_i = |L[e_i]\omega l|$

Fig. 9 Fully Lazy Lambda Lifting Rules.

duces a similar result as one by lambda hoisting, which may not be suitable for graph reduction.

3.2 Maximal Free Occurrences

For full laziness, both algorithms must detect the maximal free occurrences of subexpressions to abstract them. The major difference lies in the applying technique. This also derives from the difference of implementation schemes. The fully lazy lambda lifting transforms the expression

$$E = (\ldots E_1 \ldots E_2 \ldots)$$

(where E_1 and E_2 are all the maximal free occurrences of subexpressions in E) into the expression

$$(\textbf{fn } x_1\ x_2\text{: } (\ldots x_1 \ldots x_2 \ldots))E_1\ E_2. \tag{3}$$

At the final stage, it compiles (**fn** x_1 x_2: (. . . x_1 . . . x_2 . . .)) into a supercombinator, say Ψ, and whole expression is replaced by $\Psi\ E_1\ E_2$.

In lambda hoisting, the result $E[x/e]$ of the reduction of an application ((**fn** x: E)e) is equivalent to the expression (E **whererec** $x=e$). Hence, we can proceed to transform the expression (3) into

$$(\ldots x_1 \ldots x_2 \ldots) \textbf{ whererec } x_1 = E_1 \textbf{ and } x_2 = E_2.$$

After this, the maximal free occurrences of subexpressions can be treated as if they were originally declared in a **whererec**-clause.

In case that the maximal free occurrence of subexpression E_1 in (. . . E_1 . . .) is a variable, even if the lambda hoisting transforms (. . . E_1 . . .) into

$$(\ldots x_1 \ldots) \textbf{ whererec } x_1 = E_1,$$

when (. . . x_1 . . .) is evaluated, the whole environment becomes

$$\textbf{whererec } x_1 = E_1 \textbf{ and } \ldots \textbf{ and } E_1 = E_2 \textbf{ and } \ldots$$

Therefore it is redundant to replace the variable E_1 with a fresh identifier x_1. Thus lambda hoisting does not abstract maximal free occurrences of variables. This is one of the reasons that the lambda hoisting treats maximal free occurrences of combinations rather than expressions.

4. Transformation

We now redefine the rules for fully lazy lambda lifting by transforming the lambda hoisting rules taking account of the differences described in the previous sections. We first divide the lambda hoisting rules into two sets of rules. The first is the floating rules shown in Fig. 5 which float out the local definitions as high as possible, and the second, the abstracting rules shown in Fig. 6 which detect the maximal free occurrences of combinations and abstract them using local definitions. Note that original rules are equivalent to

$$\langle \mu^*, \omega^*, e^* \rangle = A[e' \textbf{ whererec } \mu' 0]\mu_\phi \omega_\phi 0$$

$$\text{where } \langle \mu', \omega', e' \rangle = F[e]\mu_\phi \omega_\phi 0.$$

Then we revise each set of rules to match fully lazy lambda lifting. They are shown in Fig. 7 and Fig. 8. Fully lazy lambda lifting floats out local definitions separately for each constituent of combinations and each body of local definitions. Therefore, μ is not passed as argument. In addition, new **whererec**-clauses appear as return values of expression in Fig. 7, because we must not float out the local definitions further than necessity as mentioned in the latter part of Section 3.1.

Fully lazy lambda lifting transforms **fn**-expressions into supercombinators. Hence we need to introduce σ to accumulate those definitions. In defining supercombinators, we decide the order of parameters according to Hughes [2].

Fully lazy lambda lifting abstracts a maximal free occurrence of a single variable. Thus it would occur that several parameters represent a same variable without checking it. So it behooves us to eliminate the redundant introduction of parameters.

Finally, we can combine the revised rules for floating and abstracting operations to make the fully lazy lambda lifting rules shown in Fig. 9 which uses two en-

vironments, μ and ν, to treat the declarations of local definitions and those of the maximal free occurrences of expressions separately.

5. Conclusions

We have shown that the two algorithms are very similar in that their basic operations divide into two phases. This idea is very similar to Peyton Jones [6], while its purpose is different. The local definitions are floated out in the first one, and the maximal free occurrences of subexpressions are detected and treated specially in the second one. The differences of the algorithms come from the difference of implementation schemes.

The difference in the floating out operation is due to the fact that fully lazy lambda lifting implements a local definitions as a graph, while lambda hoisting does it as a closure.

Another difference has been observed. Fully lazy lambda lifting abstracts all the maximal free occurrences of expressions including single variables to transform **fn**-expressions into supercombinators. Since lambda hoisting is based on environment model implementation, it is unnecessary to abstract maximal free occurrences of variables.

In addition, we have shown that it is possible to construct the fully lazy lambda lifting rules by transforming those for the lambda hoisting step by step. This means that there exist other feasible algorithms between the algorithms. More generally, we may say that there exist other feasible models between the graph reduction one and the environment one.

There is a pure graph reduction model such as Turner [8]'s implementation technique. On the other hand, there is an environment model such as SECD machine. The suitable model for fully lazy lambda lifting is an intermediate one placed between these models. The recent implementations of graph reduction models try to suppress construction of graphs (Augustsson [1], Peyton Jones [5]). They use graphs to represent local definitions and frames which may be thought as the environment, to execute the compiled supercombinators. Therefore, we conclude that we are free to adopt the translation algorithm according to the actual implementation scheme on the target machine.

References

1. Augustsson, L. and Johnsson, T. Parallel Graph Reduction with the $\langle v, G \rangle$-machine, *Proceedings of the 1989 Conference on Functional Programming Language and Computer Architecture* (1989), 202–213.
2. Hughes, R. J. M. Super-combinators: A New Implementation Method for Applicative Languages, *Proceedings of 1982 ACM Symposium on Lisp and Functional Programming* (1982), 1–10.
3. Johnsson, T. Lambda-Lifting: Transformation Programs to Recursive Equations, *Lecture Notes in Computer Science 201*, Springer-Verlag (1985), 190–203.
4. Peyton Jones, S. L. *The Implementation of Functional Programming Languages*, Prentice-Hall International, 1987.
5. Peyton Jones, S. L. The Spineless Tagless G-Machine, *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture* (1989), 184–201.
6. Peyton Jones, S. L. and Lester, D. R. A Modular, Fully Lazy Lambda Lifter in Haskell, *Software-Practice and Experience*, **21**, 5 (1991), 479–506.
7. Takeichi, M. Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs, *New Generation Computing*, **5** (1988), 377–391.
8. Turner, D. A. A New Implementation Technique for Applicative Languages, *Software-Practice and Experience*, **9** (1979), 31–49.

(Received February 3, 1992)