C8-2

Relation between Lambda Hoisting and Fully Lazy Lambda Lifting

Keiichi Kaneko and Masato Takeichi
Faculty of Engineering, the University of Tokyo

Abstract

In this paper, we clarify essential differences of the lambda hoisting (Takeichi[5]) and the fully lazy lambda lifting (Peyton Jones[4]) algorithms for translating functional programs into fully lazy ones. These algorithms look alike in that both of them float out local definitions, and extract maximal free occurrences of subexpressions to achieve full laziness. However, such operations differ slightly depending on their underlying implementation schemes.

We first define a functional language to explain the algorithms and then discuss the differences between the two algorithms. Finally, we claim that both algorithms are continuous by transforming the lambda hoisting rules into those for fully lazy lambda lifting.

1 Introduction

For example, let us consider a lambda expression:

$$\lambda x.\mathrm{I}(\lambda y.y\ \alpha\ (+\ x\ 1)\ \textbf{whererec}\ \alpha = +\ x\ 2)$$
$$(\mathrm{K}(\lambda z.*\ z\ \beta\ \textbf{whererec}\ \beta = +\ x\ 3))$$

where I and K are combinators. We can transform this expression using fully lazy lambda lifting into a supercombinator Ψ defined as:

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

using auxiliary supercombinators Φ_1 and Φ_2 defined as:

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

On the other hand, if we transform the expression using lambda hoisting, the following lambda expression is obtained:

$$\lambda x.\mathrm{I}(\lambda y.y\ \alpha\ \gamma)(\mathrm{K}(\lambda z.*\ z\ \beta))$$
$$\textbf{whererec}\ \alpha = +\ x\ 2\ \textbf{and}\ \beta = +\ x\ 3\ \textbf{and}\ \gamma = +\ x\ 1. \qquad (2)$$

Comparing (1) and (2), we find that a supercombinator corresponds to a lambda expression, respectively. (2) differs from (1) in two points:

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

In the rest of this paper, we investigate the reason why there are these differences.

2 Preliminaries

2.1 A Simple Functional Language

We introduce a simple functional language to provide a common means for describing the algorithms. Figure 1 shows a simple 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 virtually any local definition with **whererec**, but we can not 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 of our assumption, functional programs written in a language with more generous 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 Figure 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 are calculated. We can determine 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 every subexpression depends. Figure 3 shows the assignment rules without details. See Takeichi[5] for them.

Followings are strict definitions 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) 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 $0 \leq |L[e_0]\omega l| < l, 0 \leq |L[e_1]\omega l| < l$, 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 $(e^*\ \textbf{whererec}\ \mu^* 0)$ in the fully lazy normal form by the lambda hoisting rules shown in Figure 4:

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

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.

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

$\mathbf{fn}\ x : e^*$,

e^* $\mathbf{whererec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = e_n$, 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 $\mathbf{fn}\ x_1 x_2 \ldots x_n : e'$, and e' does not contain any lambda abstraction which is not a supercombinator.

The algorithm for floating out 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 has this level number.
3 If the definition appears in the function position of an application, it is floated out until it does not.

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

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

to get the hoisted expression

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

The environment is updated only once when the expression is evaluated. In an actual implementation, each local definition is represented by a closure which occupies a little space. So even if the variables x and y are not used in evaluation, unnecessary memory consumption is very little. 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. Therefore, if we collect local definitions in the following expression after the manner of lambda hoisting

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

we get the too lifted expression

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

When we evaluate this expression, 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.

3.2 Maximal Free Occurrences

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

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

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

$$(\mathbf{fn}\ x_1\ x_2 : (\ldots x_1 \ldots x_2 \ldots))E_1 E_2. \qquad (3)$$

At the final stage, it compiles $\mathbf{fn}\ x_1\ x_2 : (\ldots x_1 \ldots x_2 \ldots)$ 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 $((\mathbf{fn}\ x : E)\ e)$ is equivalent to the expression $(E\ \mathbf{whererec}\ x = e)$. Hence, we can proceed to transform the expression (3) into

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

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 v in $(\ldots v \ldots)$ is a variable, even if lambda hoisting transforms $(\ldots v \ldots)$ into

$$(\ldots x \ldots)\ \mathbf{whererec}\ x = v,$$

when $(\ldots x \ldots)$ is evaluated, the whole environment becomes

$$\mathbf{whererec}\ x = v\ \mathbf{and}\ \ldots\ \mathbf{and}\ v = E\ \mathbf{and}\ \ldots.$$

Therefore it is redundant to replace the variable v with a fresh identifier x. 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

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

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

Then we revise each set of rules to match fully lazy lambda lifting. They are shown in Figure 7 and Figure 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 Figure 7, because we should not float out the local definitions further than necessity.

And fully lazy lambda lifting transforms fn-expressions into supercombinators. Hence we 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 Figure 9 which uses two environments, μ and ν, to treat the declarations of local definitions and those of the maximal free occurrences of expressions separately.

5 Conclusions

In this paper, we have shown the followings. Both algorithms are very similar in the point that their basic operations consist of the floating out the local definitions and the treatment for maximal free occurrences of subexpressions. Differences in each operation derive from the difference of implementation schemes.

In addition, we have shown that we can construct the fully lazy lambda lifting rules by transforming those for the lambda hoisting step by step. This means that the transition between the algorithms are continuous. More generally, we can induce that so is the transition between the graph reduction model and the environment model.

There is a pure graph reduction model (Turner[6]). On the other hand, there is an environment model such as lambda hoisting. And fully lazy lambda lifting is an intermediate model of them. Its implementations recently use graphs to represent local definitions and frames, which can be thought as the environment, to execute the compiled supercombinators. Therefore, we should adopt the model according to the actual implementation on the target machine.

References

[1] Augustsson, L. and T. Johnsson: "Pararell Graph Reduction with the $\langle \nu, G \rangle$-machine," *Proceedings of the 1989 Conference on Functional Programming Language and Computer Architecture*, pp. 202–213, 1989.

[2] Hughes, R. J. M.: "Super-combinators: A New Implementation Method for Applicative Languages," *Proceedings of 1982 ACM Symposium on Lisp and Functional Programming*, pp. 1–10, 1982.

[3] Johnsson, T.: "Lambda-Lifting: Transformation Programs to Recursive Equations," *Lecture Notes in Computer Science 201*, Springer-Verlag, pp. 190–203, 1985.

[4] Peyton Jones, Simon L.: "*The Implementation of Functional Programming Languages*," Prentice-Hall Inernational, 1987.

[5] Takeichi, M.: "Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs," *New Generation Computing*, Vol. 5, pp. 377–391, 1988.

[6] Turner, D. A.: "A New Implementation Technique for Applicative Languages," *Software–Practice and Experience*, No. 9, pp. 31–49, 1979.

Syntactic Domains

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

Abstract Syntax

$e ::= b \mid x \mid ee \mid \mathbf{fn}\ x : e \mid e\ \mathbf{whererec}\ x = e\ \mathbf{and} \ldots \mathbf{and}\ x = e$

Figure 1. Specification of Our Functional Language

Syntax

$e ::= e' \mid e'\ \mathbf{whererec}\ x = e'\ \mathbf{and} \ldots \mathbf{and}\ x = e'$
$e' ::= b \mid x \mid e'e' \mid \mathbf{fn}\ x : e$

Context Condition

e contains no free occurrence of compound expressions.

Figure 2 Fully Lazy Normal Form

Level Numbers

$l \in \mathbf{N}$

Environment for Level Numbers

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

Assignment Rules

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

$L[b]\omega l = \{0\}$
$L[x]\omega l = \{0\} \cup \{\omega[x]\}$
$L[e_0 e_1]\omega l = L[e_0]\omega l \cup L[e_1]\omega l$
$L[\mathbf{fn}\ x : e_0]\omega l = L[e_0](\omega + \langle x \to l+1 \rangle)(l+1) - \{l+1\}$
$L[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = e_n]\omega l = L[e_0]\omega' l$
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

Operator + stands for the disjoint sum.
For any domain $\mathbf{X}$, $\mathbf{X}_+ = \mathbf{X} + \{\mathbf{err}\}$.
For an environment ω, $\omega + \langle x \to l \rangle$ denotes
$\lambda y.\mathbf{if}\ x = y\ \mathbf{then}\ l\ \mathbf{else}\ \omega[y]$.

Initial Environment

For pre-defined identifiers x, ω_ϕ satisfies $\omega_\phi[x] = 0$.

Figure 3 Rules for Assigning Level Numbers

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

Notations

Tuples in $[\mathbf{M} \times \mathbf{L} \times \mathbf{Exp}]$ are written as $\langle \mu, \omega, e \rangle$.
Syntactic elements are quoted by $\lceil$ and $\rceil$.
For a declaration set μ, $\mu + \langle k \to \nu \rangle$ denotes
$\lambda j.\ \mathbf{if}\ j = k\ \mathbf{then}\ \nu\ \mathbf{else}\ \mu j$
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

For any $l \in \mathbf{N}$, μ_ϕ satisfies $\mu_\phi l = \{\}$.

Figure 4 Lambda Hoisting Rules

Floating Rules

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

$F[b]\mu\omega l = \langle \mu, \omega, \lceil b \rceil \rangle$
$F[x]\mu\omega l = \langle \mu, \omega, \lceil x \rceil \rangle$
$F[e_0 e_1]\mu\omega l = \langle \mu^*, \omega^*, e^* \rangle$
let $\langle \mu', \omega', e_0' \rangle = F[e_0]\mu\omega l$ in
let $\langle \mu^*, \omega^*, e_1' \rangle = F[e_1]\mu'\omega' l$ in
$e^* = \lceil (e_0' e_1') \rceil$
$F[\mathbf{fn}\ x : e_0]\mu\omega l = \langle \mu^*, \omega^*, \lceil \mathbf{fn}\ x : e_0^* \rceil \rangle$
let $\langle \mu^*, \omega^*, e_0' \rangle = F[e_0](\mu + \langle l+1 \to \{\} \rangle)(\omega + \langle x \to l+1 \rangle)(l+1)$ in
$e_0^* = \lceil e_0'\ \mathbf{whererec}\ \mu^*(l+1) \rceil$
$F[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = e_n]\mu\omega l = F[e_0]\mu_n\omega_n l$
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 $\mu_0 = \mu$, $\omega_0 = \omega$

Figure 5 Floating Rules for Local Definitions

Abstracting Rules

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

$A[b]\mu\omega l = \langle \mu, \omega, \lceil b \rceil \rangle$
$A[x]\mu\omega l = \langle \mu, \omega, \lceil x \rceil \rangle$
$A[e_0 e_1]\mu\omega l = \langle \mu^*, \omega^*, e^* \rangle$
  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' $(i = 0, 1)$ 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 $e^* = \lceil (e_0' e_1') \rceil$
$A[\mathbf{fn}\ x :\ e_0]\mu\omega l = \langle \mu^*, \omega^*, \lceil \mathbf{fn}\ x :\ e_0^* \rceil \rangle$
  let $\langle \mu', \omega', e_0' \rangle = A[e_0](\mu + \langle l+1 \to \{\} \rangle)(\omega + \langle x \to l+1 \rangle)(l+1)$ 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 $e^* = \lceil e_0'\ \mathbf{whererec}\ \mu'(l+1) \rceil$
$A[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\mu\omega l = \langle \mu^*, \omega^*, e_0^* \rangle$
  let $\langle \mu^*, \omega^*, e_0' \rangle = A[e_0]\mu_n'\omega_n' l$
    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 $\mu_0' = \mu$, $\omega_0' = \omega$
    in $e_0^* = \lceil e_0'\ \mathbf{whererec}\ x_1 = e_1'\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n'\ \mathbf{and}\ \mu^* l \rceil$

Figure 6 Abstracting Rules for MFOCs

Floating Rules

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

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

Notation

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

Figure 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

$A' : \mathbf{Exp} \to \mathbf{M} \to \mathbf{L} \to \mathbf{N} \to \mathbf{S} \to [\mathbf{M} \times \mathbf{L} \times \mathbf{Exp} \times \mathbf{S}]$

$A'[b]\mu\omega l\sigma = \langle \mu, \omega, \lceil b \rceil, \sigma \rangle$
$A'[x]\mu\omega l\sigma = \langle \mu, \omega, \lceil x \rceil, \sigma \rangle$
$A'[e_0 e_1]\mu\omega l\sigma = \langle \mu^*, \omega^*, e^*, \sigma^* \rangle$
  let $\langle \mu', \omega', e_0', \sigma' \rangle = A'[e_0]\mu\omega l\sigma$ in
    let $\langle \mu'', \omega'', e_1', \sigma^* \rangle = A'[e_1]\mu'\omega' l\sigma'$ in
      if e_i' $(i = 0, 1)$ 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 $e^* = \lceil (x' e_1') \rceil$ or $e^* = \lceil (e_0' x') \rceil$
          for $i = 0, 1$, respectively,
        else $\mu^* = \mu'' + \langle k \to \mu'' k \cup \lceil x' = e_i' \rceil \rangle$, $\omega^* = \omega'' + \langle x' \to k \rangle$,
          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 $e^* = \lceil (e_0' e_1') \rceil$
$A'[\mathbf{fn}\ x :\ e_0]\mu\omega l\sigma = A'[e_0^*]\mu\omega^* l\sigma^*$
  let $\langle \mu', \omega', e_0', \sigma' \rangle = A'[e_0]\{\}(\omega + \langle x \to l+1 \rangle)(l+1)\sigma$ in
    if e_0' is an MFOE w.r.t. ω' and $l+1$,
      $\omega^* = \omega' + \langle \Phi \to 0 \rangle$, $e_0^* = \lceil \Phi e_0' \rceil$ and $\sigma^* = \sigma' \cup \{\lceil \Phi x' x = x' \rceil\}$
      where Φ and x' are fresh identifiers
    else $\omega^* = \omega' + \langle \Phi \to 0 \rangle$, $e_0^* = \lceil \Phi e_{01}' \ldots e_{0n_0}' \ldots e_{l1}' \ldots e_{ln_l}' \rceil$
      and $\sigma^* = \sigma' \cup \lceil \Phi x_{01}' \ldots x_{0n_1}' \ldots x_{0l}' \ldots x_{0n_l}' x = e_0' \rceil$
      where Φ is a fresh identifier
      and $\mu' k$ is $\{\lceil x_{k1}' = e_{k1}' \rceil, \ldots, \lceil x_{kn_k}' = e_{kn_k}' \rceil\}$
$A'[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\mu\omega l\sigma$
$= \langle \mu^*, \omega^*, e_0^*, \sigma^* \rangle$
  let $\langle \mu^*, \omega^*, e_0', \sigma^* \rangle = A'[e_0]\mu_n'\omega_n' l\sigma_n'$ in
    $e_0^* = \lceil e_0'\ \mathbf{whererec}\ x_1 = e_1'\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n' \rceil$
    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 $\mu_0' = \mu$, $\omega_0' = \omega$, $\sigma_0' = \sigma$

Figure 8 Revised Abstracting Rules for MFOEs

Lifting Rules

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

$L'[b]\nu\omega l\sigma = \langle \mu_\phi, \nu, \omega, \lceil b \rceil, \sigma \rangle$
$L'[x]\nu\omega l\sigma = \langle \mu_\varphi, \nu, \omega, \lceil x \rceil, \sigma \rangle$
$L'[e_0 e_1]\nu\omega l\sigma = \langle \mu^*, \nu^*, \omega^*, e^*, \sigma^* \rangle$
  let $\langle \mu'', \nu'', \omega'', e_1', \sigma'' \rangle = L'[e_1]\nu'\omega' l\sigma'$
    where $\langle \mu', \nu', \omega', e_0', \sigma' \rangle = L'[e_0]\nu\omega l\sigma$ in
    $\mu^* = \mu' + (\mu'' + \langle l \to \{\} \rangle)$,
    if e_i' $(i = 0, 1)$ 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 $e^* = \lceil (x'(e_1'\ \mathbf{whererec}\ \mu'' l)) \rceil$
        or $e^* = \lceil (e_0' x') \rceil$ for $i = 0, 1$, respectively
      else $\nu^* = \nu'' + \langle k \to \nu'' k \cup \{\lceil x' = e_i' \rceil\} \rangle$, $\omega^* = \omega'' + \langle x' \to k \rangle$,
        and $e^* = \lceil (x'(e_1'\ \mathbf{whererec}\ \mu'' l)) \rceil$ 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 $e^* = \lceil (e_0'(e_1'\ \mathbf{whererec}\ \mu'' l)) \rceil$
$L'[\mathbf{fn}\ x :\ e_0]\nu\omega l\sigma = \langle \mu^*, \nu^*, \omega^*, e_0^*, \sigma \rangle$
  let $\langle \mu', \nu', \omega', e_0', \sigma' \rangle = L'[e_0]\nu_\phi(\omega + \langle x \to l+1 \rangle)(l+1)\sigma$ in
    let $\langle \mu'', \nu^*, \omega^*, e_0^*, \sigma^* \rangle = L'[e_0']\nu_\phi\omega'' l\sigma''$ in $\mu^* = \mu' + \mu''$
    where
      if e_0' is an MFOE w.r.t. ω' and $l+1$,
        $\omega'' = \omega' + \langle \Phi \to 0 \rangle$, $e_0'' = \lceil (\Phi e_0') \rceil$,
        and $\sigma'' = \sigma' \cup \{\lceil \Phi x' x = x' \rceil\}$
      else $\omega'' = \omega' + \langle \Phi \to 0 \rangle$, $e_0'' = \lceil (\Phi e_{01}' \ldots e_{0n_0}' \ldots e_{l1}' \ldots e_{ln_l}') \rceil$,
        and $\sigma'' = \sigma' \cup \{\lceil \Phi x_{01}' \ldots x_{0n_0}' \ldots x_{l1}' \ldots x_{ln_l}'$
          $= e_0'\ \mathbf{whererec}\ \mu'(l+1) \rceil\}$
        where Φ is a fresh identifier
        and νk is $\{\lceil x_{k1}' = e_{k1}' \rceil \ldots \lceil x_{kn_k}' = e_{kn_k}' \rceil\}$
$L'[e_0\ \mathbf{whererec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n]\nu\omega l\sigma$
$= \langle \mu^*, \nu^*, \omega^*, e_0^*, \sigma^* \rangle$
  let $\langle \mu_0', \nu^*, \omega^*, e_0^*, \sigma^* \rangle = L'[e_0]\nu_n'\omega_n' l\sigma_n'$ in
    $\mu^* = \mu_0' + \sum_{i=1}^{n} (\mu_i' + \langle k_i \to \{\lceil x_i = e_i'\ \mathbf{whererec}\ \mu_i' k \rceil\} \rangle)$
    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 $\mu_0' = \mu, \omega_0' = \omega, \sigma_0' = \sigma, k_i = |L[e_i]\omega l|$

Figure 9 Fully Lazy Lambda Lifting Rules