C－3－4

Lambda-hoisting: a transformation technique for fully lazy evaluation of functional programs (Extended Abstract)

Masato Takeichi

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

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 algorithm is presented with remarks on similar techniques.

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]. 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 produced during compilation and dependent on the source program. 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.

In this paper we present a technique called *lambda-hoisting* for transforming programs into ones convenient for fully lazy evaluation. Evaluation of the transformed program in the ordinary lazy way [1,2] results in fully lazy evaluation of the original program. It enables one to implement full laziness by means of an ordinary lazy evaluator.

We use a simple functional language shown in Figure 1.

2. Lambda-hoisting Algorithm

Lambda-hoisting is similar to the super-combinator method. Consider a function *el* 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[1]:

el = fn n:
  {fn s: *IF* (= n 1)(*HEAD* s)(*el*(− n 1)(*TAIL* s))}

For the function *el*, we can derive a super-combinator ϕ by Hughes' method as[2]

el = ϕ (*IF* (= n 1)) (*el* (− n 1))
  where ϕ = fn a b s: 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 increasing operations for passing arguments of a super-combinator to other super-combinators.

Transformation by lambda-hoisting yields

el = fn n:
  { fn s: a (*HEAD* s) (b (*TAIL* s))
    whererec a=*IF* (= n 1) and b=*el*(− n 1) }

Thus, the resulting function contains local definitions by a whererec-clause.

2.1. 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-

[1] 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.

[2] We shall use hereafter an extended form of fn-abstraction: fn $x_1\, x_2 \cdots x_n$: e_0. It is a natural extension to fn x: e_0, while it is not equivalent to fn x_1: (fn x_2: $\cdots$ (fn x_n: e_0)). See Section 2.3.

combinator must not. The *fully lazy normal form* is a sublanguage of the language defined in Figure 1. Its syntax rules and context condition are shown in Figure 2.

The semantic function E of Figure 1 is supposed to be applied to the new syntactic category e' as well as e. Note that where-clauses disappear in the fully lazy form. Local definitions by **whererec**-clauses may appear only in the outermost expression, i.e., the program, or in the body of fn-abstraction.

We shall develop an algorithm for transforming any expressions into expressions of the fully lazy 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.

2.2. 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 Figure 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 $x' \in \mathbf{Ide}$, $(\pi + <x \rightarrow x'>)\,[y] \neq x'$.

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 the program is preserved through the transformation by the rules in Figure 3.

2.3. Computing lexical levels

The main part of lambda-hoisting is to identify maximal free occurrences of expressions (See Section 2.4). 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 $\bar{l} = \{l_1, l_2, \cdots, l_n\}$ by

$$|\bar{l}| = \max\{l_1, l_2, \cdots, l_n\}$$

The rules for assigning the set of level numbers to the expression are shown in Figure 4. Since lexical levels are computed for expressions transformed by the rules in Section 2.2, the rule for **where**-abstraction is not used in our lambda-hoisting algorithm, though it is included in Figure 4.

The rule for fn-abstractions is worth noting. Assume that $\mathbf{fn}\ x\colon e_0$ appears in the context $\omega \in \mathrm{L}$ of the level l. If e_0 contains x and no other variables, we get $|L[\mathbf{fn}\ x\colon e_0]\omega l| = 0$ from the rule. Hence the combinator has the level 0 in any context and at any level. For example, an occurrence of a combinator

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

is considered as a constant[3].

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

$$\omega' = \omega + <x_1 \rightarrow |L[e_1]\omega' l|> + \cdots + <x_n \rightarrow |L[e_n]\omega' l|>$$

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 $i = 1, \cdots, n$;
repeat
  $l_i := l_i'$ for $i = 1, \cdots, n$;
  $\omega' := \omega + <x_1 \rightarrow l_1> + \cdots + <x_n \rightarrow l_n>$;
  $l_i' := |L[e_i]\omega' l|$ for $i = 1, \cdots, n$;
until $\{ l_i = l_i'$ for every $i = 1, \cdots, n \}$

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

2.4. 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 2.2.

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 $\omega \in \mathrm{L}$ 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 $\omega \in \mathrm{L}$ and $l \in \mathbf{N}$ is called *maximal*, if either of following conditions holds.

---

[3] In the extended fn-binding $\mathbf{fn}\ x_1 \cdots x_n$, we consider that variables $x_1, \cdots, x_n$ are of the same level, say $(l+1)$ if the fn-abstraction appears at level l.

(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 $\cdots$ and $x_n = e_n$

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

Rules for lambda-hoisting are shown in Figure 5. In short, maximal free occurrences 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

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

For example,

fn x: { fn y: (+ (− x 1) y) }

is transformed into

fn x: { fn y: (z y) } whererec z=(+ (− x 1))

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

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

$$E[e^*]\rho_0 = E[e]\rho_0$$

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.

3. 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 " $\cdots$ for $i=1, \cdots, n$". We have a program written in Lisp for transforming Lispkit Lisp [4] programs into fully lazy normal forms. 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, and Melcom Cosmo. As the task of the code generator is a simple macro processing driven by a table, it is easy to generate code for other machines.

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.

References

1. Friedman, D. P., Wise, D. S.: CONS should not evaluate its arguments. *Proc. 3rd International Colloquium on Automata, Languages and Programming*, 257-284 (1976)
2. Henderson, P., Morris, J. M.: A lazy evaluator. *Proc. 3rd Symp. on POPL*, 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*, 1-10 (1982)
6. Hughes, R. J. M.: *The design 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*, 58-69 (1984)
8. Johnsson, T: Lambda-lifting: Transforming Programs to Recursive Equations. *LNCS 201*, 190-203, Springer-Verlag, 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*, 11-20 (1982)
10. Landin, P. J.: The next 700 programming languages, *Comm. ACM*, 157-164 (1966)
11. Takeichi, M.: Evaluation of combinator expressions. *Proc. 1st JSSST Annual Conference*, 213-222 (1984). In Japanese.
12. Takeichi, M.: An Alternative Scheme for Evaluating Combinator Expressions, *Journal of Information Processing*, 246-253 (1985)
13. Takeichi, M.: A Functional Machine for Fully Lazy Evaluation (Extended Abstract), *Proc. RIKEN Symp. on Functional Programming*, 54-60 (1986)
14. Takeichi, M.: Partial Parametrization Eliminates Multiple Traversals of Data Structures. To appear in *Acta Informatica*.
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*, 39-49 (1979)
17. Turner, D. A.: *Aspects of the implementation of programming languages*. D.Phil. thesis. Oxford University, 1981.
18. Turner, D. A.: Miranda: A non-strict functional language with polymorphic types, *LNCS 201*, 1-16, Springer-Verlag, 1985.

Figure 1. Denotational Specification of a Simple Functional Language

Syntactic domains

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

Abstract syntax

$e ::= b \mid x \mid e\ e \mid \mathbf{fn}\ x\mathbf{:}\ e \mid$
$e\ \mathbf{where}\ x\mathbf{=}e\ \mathbf{and}\ \cdots\ \mathbf{and}\ x\mathbf{=}e \mid$
$e\ \mathbf{whererec}\ x\mathbf{=}e\ \mathbf{and}\ \cdots\ \mathbf{and}\ x\mathbf{=}e$

Semantic domains

$\mathbf{B}$	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

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

$E[b]\rho = B[b]$
$E[x]\rho = \rho[x]$
$E[e_0\ e_1]\rho = (E[e_0]\rho)(E[e_1]\rho)$
$E[\mathbf{fn}\ x\mathbf{:}\ e_0]\rho = \lambda\delta.E[e_0](\rho + \langle x \to \delta \rangle)$
$E[e_0\ \mathbf{where}\ x_1\mathbf{=}e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n\mathbf{=}e_n]\rho = E[e_0]\rho'$
where $\rho' = \rho + \langle x_1 \to E[e_1]\rho \rangle + \cdots + \langle x_n \to E[e_n]\rho \rangle$
$E[e_0\ \mathbf{whererec}\ x_1\mathbf{=}e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n\mathbf{=}e_n]\rho = E[e_0]\rho'$
where $\rho' = \rho + \langle x_1 \to E[e_1]\rho' \rangle + \cdots + \langle x_n \to E[e_n]\rho' \rangle$

Notations

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

Initial Environment

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

Figure 2. Fully Lazy Normal Form

Syntax

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

Context condition

Expression e does not contain any free occurrences of combinations.

Figure 3. Rules for Renaming Identifiers and Rewriting where-clauses

Environment for renaming

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

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

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

Notation

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

Initial Environment

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

Figure 4. Rules for Assigning Level Numbers to Expressions

Level numbers

$l \in \mathbf{N}$

Environment for level numbers

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

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

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

Notation

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

Initial Environment

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

Figure 5. Lambda-hoisting Rules for Expressions of the Fully Lazy Form

Level numbers

$l \in \mathbf{N}$

Environment for level numbers

$\omega \in \mathbf{L}$

Declarations of maximal free occurrences of combinations

$\mu \in \mathbf{M} = [\ \mathbf{N} \to 2^{\mathbf{Dec}}\ ]$
$d \in \mathbf{Dec}$ declarations
$d ::= x\mathbf{=}e$

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

$H[b]\mu\omega l = \langle \mu, \omega, [b] \rangle$
$H[x]\mu\omega l = \langle \mu, \omega, [x] \rangle$
$H[e_0\ e_1]\mu\omega l = \langle \mu^*, \omega^*, e^* \rangle$
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,
$\mu^* = \mu'' + \langle k \to \mu''k \cup [x' = e_i] \rangle$, $\omega^* = \omega'' + \langle x' \to k \rangle$,
and $e^* = [(x'\ e_1')]$ or $e^* = [(e_0'\ x')]$ for $i = 0,1$, respectively,
where $k = |L[e_i]\omega'' l|$ and x' is a fresh identifier
else $\mu^* = \mu''$, $\omega^* = \omega''$ and $e^* = (e_0'\ e_1')$
$H[\mathbf{fn}\ x\mathbf{:}\ e_0]\mu\omega l = \langle \mu^*, \omega^*, [\mathbf{fn}\ x\mathbf{:}\ e_0^*] \rangle$
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,
$\mu^* = \mu' + \langle k \to \mu' k \cup [x' = e_0'] \rangle$, $\omega^* = \omega' + \langle x' \to k \rangle$, and $e^* = [x']$
where $k = |L[e_0']\omega' l|$ and x' is a fresh identifier
else $\mu^* = \mu'$, $\omega^* = \omega'$, and $e^* = [e_0'\ \mathbf{whererec}\ \mu'(l+1)]$
$H[e_0\ \mathbf{whererec}\ x_1\mathbf{=}e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n\mathbf{=}e_n]\mu\omega l = H[e_0]\mu_n\omega_n l$
where $\mu_i = \mu_i' + \langle k \to \mu_i' \cup [x_i = e_i'] \rangle$ 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 $i = 1, \cdots, n$, 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, $\mu + \langle k \to \sigma \rangle$ denotes
$\lambda l.\mathbf{if}\ j\mathbf{=}k\ \mathbf{then}\ \sigma\ \mathbf{else}\ \mu j$
If $\mu l = \{\lceil x_1 = e_1 \rceil, \cdots, \lceil x_n = e_n \rceil\}$, $\lceil e_0\ \mathbf{whererec}\ \mu l \rceil$ denotes
$\lceil e_0\ \mathbf{whererec}\ x_1\mathbf{=}e_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ x_n\mathbf{=}e_n \rceil$

Initial set of declarations

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

Figure 6. An Example of Lambda-hoisting

Source program:

$\mathbf{fn}\ x$:
$\{\ \xi\ FORK\ (\mathbf{fn}\ u\mathbf{:}\ TIP\ (\xi\ MIN\ l))$
$\mathbf{where}\ \xi = btree\ x$
$\mathbf{whererec}$
$btree = \mathbf{fn}\ x\mathbf{:}$
$\{\mathbf{fn}\ g\ f\mathbf{:}$
$IF\ (ISTIP\ x)\ (f\ (TIPVAL\ x))$
$(g\ (btree\ (LEFT\ x)\ g\ f)\ (btree\ (RIGHT\ x)\ g\ f))\ \}\ \}$

Fully lazy normal form:

$\{\ \mathbf{fn}\ x'\mathbf{:}\ \{\ \xi'\ FORK\ (\mathbf{fn}\ u'\mathbf{:}\ \alpha)\ \}\ \mathbf{whererec}\ \xi' = btree'\ x'\ \mathbf{and}\ \alpha = TIP\ (\xi'\ MIN\ l)\ \}$
$\mathbf{whererec}$
$btree' = \mathbf{fn}\ x''\mathbf{:}\ \{\ \mathbf{fn}\ g'\ f'\mathbf{:}\ \beta_2\ (f'\ \beta_1)\ (g'\ (\beta_3\ g'\ f')\ (\beta_4\ g'\ f'))\ \}$
$\mathbf{whererec}\ \beta_1 = TIPVAL\ x''\ \mathbf{and}\ \beta_2 = IF\ (ISTIP\ x'')$
$\mathbf{and}\ \beta_3 = btree'\ (LEFT\ x'')\ \mathbf{and}\ \beta_4 = btree'\ (RIGHT\ x'')$