# Compiling Lazy Functional Programs Based on the Spineless Tagless G-machine for the Java Virtual Machine

Kwanghoon Choi, Hyun-il Lim, and Taisook Han

Department of Electrical Engineering & Computer Science,
Korea Advanced Institute of Science and Technology, Taejon, Korea
{khchoi,hilim,han}@cs.kaist.ac.kr

**Abstract.** A systematic method of compiling lazy functional programs based on the Spineless Tagless G-machine (STGM) is presented for the Java Virtual Machine (JVM). A new specification of the STGM, which consists of a compiler and a reduction machine, is presented; the compiler translates a program in the STG language, which is the source language for the STGM, into a program in an intermediate language called L-code, and our reduction machine reduces the L-code program into an answer. With our representation for the reduction machine by the Java language, an L-code program is translated into a Java program simulating the reduction machine.

The translated Java programs also run at a reasonable execution speed. Our experiment shows that execution times of translated benchmarks are competitive compared with those in a traditional Haskell interpreter, Hugs, particularly when Glasgow Haskell compiler's STG-level optimizations are applied.

## 1 Introduction

The motivation to compile functional languages for the Java Virtual Machine (JVM) [4] comes from the mobility of being able to run the same compiled code on any machine with a JVM, and the potential benefits of interlanguage working with Java. For implementors, the fact that every JVM is equipped with a built-in garbage collector may draw their attention. As a method of functional language implementations, this subject may be an interesting exploration of the strange nexus of functional languages, byte coded virtual machines, and object oriented programming.

Our object is to develop a lazy functional language compiler based on the Spineless Tagless G-Machine (STGM) for the JVM. The reason for using the abstract machine is that it is the state of the art in lazy abstract machine and many optimizations are available for its source language, the Shared Term Graph (STG) [7][8].

There have already been two similar attempts to do this by Tullsen [9] and Vernet [10]. They have tried to translate an STG program directly into a Java

H. Kuchen and K. Ueda (Eds.): FLOPS 2001, LNCS 2024, pp. 92–107, 2001.
© Springer-Verlag Berlin Heidelberg 2001

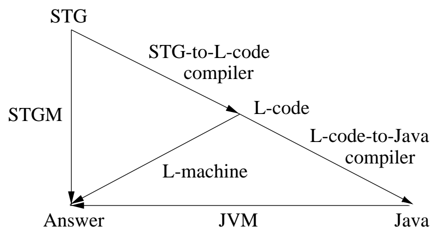


**Fig. 1.** A Two-Level Translation Scheme

program. Since the STG is simply a functional language, it is the lower level implementation decisions that make the abstract machine spineless and tagless. Hence, in the direct translations, the issues in implementing the STGM are forced to blend with the ones in the representation with Java codes. Besides, both researchers describe their translation methods informally with several examples. For some thing, like tail recursion, it is not clear how they implemented it.

In this study, we develop a two-level translation scheme: the first phase for compiling the STG language into an intermediate language called L-code (lambda code) and the second phase for representing the L-code language with the Java codes, as shown in Figure 1. The L-code is designed to specify precisely the behavior of STG programs with respect to the STGM. Each instruction of the language materializes explicitly the operations on closures, a register file, a stack, or a heap, which may be derived from the intended meaning of the STG language, so it can be directly mapped onto some Java statements. A basic unit of L-code instuctions is clearly identified in the structure of L-code programs, so it can be directly mapped onto exactly one class in Java. Every L-code program will reflect whole aspects necessary for the mapping. With this intermediate language, the translation issues can be separated from the representation ones, and each specific matter can be concentrated on in a modular way. For each level, a *concrete* compilation rule is given, and it is perfectly reproducible. Based on this scheme, we have developed an LFL-to-Java compiler. For five initial benchmarks, performance is very promising.

The contributions of our study are as follows:

- A systematic method of mapping the STGM onto arbitrary machines by the STG-to-L-code compiler and the reduction machine is provided (Section 2).
- The representations necessary for the mapping from L-code to Java are determined, and an L-code-to-Java compiler is presented (Section 3 and 4).
- The performance of the Java programs produced is measured (Section 5).

After surveying related works in Section 6, we conclude in Section 7.

## 2 The Spineless Tagless G-machine

In this section, the STGM is described by four components: the STG language, the L-code, an STG-to-L-code compiler, and the L-machine.

### 2.1 Source Language

The STGM treates the STG language as its source language. The syntax of our STG language is as follows:

| | | | |
|---|---|---|---|
| $x, y, z, w$ | | | variable |
| $v$ | $::=$ | $\lambda\overline{x}.e \mid c\ \overline{x}$ | value |
| $e$ | $::=$ | $x_0\ \overline{x} \mid \text{let}\ \overline{bind}\ \text{in}\ e \mid \text{case}\ e\ \text{of}\ \lambda z.alts$ | expression |
| $b$ | $::=$ | $v \mid e$ | bound expression |
| $\pi$ | $::=$ | $u \mid n$ | update flag |
| $bind$ | $::=$ | $x \stackrel{\pi}{=} b$ | binding |
| $t$ | $::=$ | $c \mid \lambda \mid \text{default}$ | tag |
| $alts$ | $::=$ | $\overline{alt}$ | alternatives |
| $alt$ | $::=$ | $c\ \overline{x} \rightarrow e \mid \text{default} \rightarrow e$ | alternative |
| $decl$ | $::=$ | $\text{T}\ \overline{c_i\ n_i}$ | type declaration |
| $prg$ | $::=$ | $\overline{decl},\ \text{let}\ \overline{bind}\ \text{in}\ main$ | program |

The notation $\overline{obj}_n$ denotes $obj_1\ ...\ obj_n$ for $n \geq 0$. All bound expressions are syntactically divided into either weak head normal form (Whnf) $v$ or non-Whnf $e$. Every binding has a update flag $\pi$ to determine whether the value of a non-Whnf is shared or not. In case expression, the $z$ is a bound variable that is bound to some resulting value of its case scrutinee $e$. The symbol $\lambda$ as a tag over lambda abstractions is distinguished from constructor tags $c$, and the symbol default is considered as a special tag for convenience. An STG program consists of a let expression and type declarations that define tags and arities of constructors. The full detail can be found in Peyton Jones's paper [7]. An example codelet is as follows:

$$\text{let}\quad s \stackrel{n}{=} \lambda f\ g\ x.\ \text{let}\ a \stackrel{u}{=} g\ x\ \text{in}\ f\ x\ a \quad \text{in}\ ...$$

### 2.2 Target Language

The L-code is determined as the target language. The L-code is a kind of higher-order assembly language in which binding relationships in the source phrase are represented using those in the $\lambda$-term, as introduced by Wand [12].

| | | | |
|---|---|---|---|
| x,y,z,w,t,s | | | register |
| $l$ | | | label |
| C ::= **let** $\overline{l = \mathsf{C}}$ **in** C | | | L-code |
| \| JMPC x | \| JMPK x y t | \| CASE t $\overline{t \rightarrow \langle l, \mathsf{x}_0\, \overline{\mathsf{x}} \rangle}$ | |
| \| GETN ($\lambda$x.C) | \| GETNT ($\lambda$x.$\lambda$t.C) | \| GETA ($\lambda\langle l, \mathsf{x}_0\, \overline{\mathsf{x}}\rangle$.C) | |
| \| PUTARG $\overline{\mathsf{x}}$ C | \| GETARG ($\lambda\overline{\mathsf{x}}$.C) | \| ARGCK $n$ $\mathsf{C}_{mp}$ C | |
| \| PUTK $\langle l, \overline{\mathsf{x}}\rangle$ C | \| GETK ($\lambda\langle l, \overline{\mathsf{x}}\rangle$.C) | \| REFK ($\lambda$x.C) | |
| \| SAVE ($\lambda$s.C) | \| DUMP C | \| RESTORE s C | |
| \| PUTC $\overline{\mathsf{x} \stackrel{\pi}{=} \langle l, \overline{\mathsf{x}}\rangle}$ C | \| GETC x ($\lambda\langle l, \overline{\mathsf{w}}\rangle$.C) | \| UPDC x $\langle l, \overline{\mathsf{w}}\rangle$ | |
| \| PUTP $\overline{\langle l, \mathsf{x}\rangle}$ ($\lambda$y.C) | \| STOP x | | |

A register is an identifier used in L-code. A label $l$ is uniquely given in **let** blocks for a sequence of L-code instructions. An L-code program for a given STG program will be a nested **let** expression. The meaning of each L-code instruction is precisely defined by the reduction machine shown in the next section.

The purpose for the introduction of L-code is to identify explicitly every operation performed in the STGM by each instruction. This enables us to show our compiler with clearly set out compilation rules as will be shown. The complete specification using L-code for the STGM is also helpful to implement our compiler accurately because it specifies precisely how the STGM is to be implemented. The STG language itself has an operational reading so that it is not impossible to translate the STG language directly into Java, but the language leaves unspecified quite a few things for the implementation of the STGM.

### 2.3 L-machine

The L-machine is defined as a set of reduction rules for L-code as shown in Figure 2, and it uses runtime structures as follows:

| | | |
|---|---|---|
| $\langle l, \rho \rangle$ | | closure |
| $\rho ::=$ | $\overline{x}$ | environment |
| $\mu ::=$ | $\mu_0 \mid \mu[\mathsf{x} \mapsto x] \mid \mu[\mathsf{x} \mapsto l] \mid \mu[\mathsf{t} \mapsto t] \mid \mu[\mathsf{s} \mapsto s]$ | register file |
| $s ::=$ | $s_0 \mid x\ s \mid \langle l, \rho \rangle\ s$ | stack |
| $h ::=$ | $h_0 \mid h[x \mapsto \langle l, \rho \rangle] \mid h[l \mapsto \mathsf{C}]$ | heap |

The reduction machine is pattern-driven; a reduction rule that matches the pattern of some current state is selected and applied. Usually, a state has the form of $\mathsf{C}\ \mu\ s\ h$. A register file $\mu$ is a mapping of registers into addresses, labels, tags, or stacks. Note that an address is represented by a variable. An environment $\rho$ is a sequence of addresses. A closure $\langle l, \rho \rangle$ consists of a label $l$ and an environment $\rho$. A stack $s$ is a sequence of addresses and closures. A heap $h$ is a mapping of addresses into closures and labels into L-codes. Here, $\mu_0$, $s_0$, and $h_0$ are an empty register file, stack, and heap respectively.

In order to give readers intuition, we give a short informal explanation about the behavior of L-machine for each instruction: the JMPC x makes a tail call to the heap-allocated closure at the address stored in x. The JMPK x y t makes a tail call to the stack-allocated closure of the label stored in x, passing the address stored in y and a tag stored in t to the closure. The CASE t $\overline{t_i \rightarrow \langle l_i, \mathsf{x}_0\, \overline{\mathsf{x}}_i \rangle}$ selects a matching alternative of $t_i$ and jumps to the instructions of the label $l_i$, passing the addresses stored in $\mathsf{x}_0$ and $\overline{\mathsf{x}}_i$. The GETN, GETNT, and GETA receive the addresses passed by JMPC, JMPK, and CASE, respectively. The PUTARG inserts the addresses of some arguments at the top of a stack and the GETARG deletes them. The ARGCK $n$ $\mathsf{C}_{mp}$ C checks the availablility of $n$ arguments on stack. Then, it jumps to C if all $n$ arguments are available, otherwise it jumps to $\mathsf{C}_{mp}$ in order to make a partial application. The PUTK, GETK, and REFK manipulate stack-allocated closures, while PUTC, GETC, UPDC, and PUTP work with heap-allocated closures. The SAVE, DUMP, and RESTORE control a stack, and the STOP x stops the reduction machine yielding an answer.

| | | |
|---|---|---|
| $(\mathbf{let}\ \overline{l_i = \mathsf{C}_i}\ \mathbf{in}\ \mathsf{C})\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu\ s\ h[\overline{l_i \mapsto \mathsf{C}_i}]$ | |
| $(\mathsf{STOP}\ \mathsf{x})\ \mu\ s\ h$ | $\Rightarrow\ (\mu(\mathsf{x}), h)$ | |
| $(\mathsf{JMPC}\ \mathsf{x})\ \mu\ s\ h$ | $\Rightarrow\ h(l)\ \mu(\mathsf{x})\ s\ h$ | $\theta(\mathsf{x}, l, \rho)^1$ |
| $(\mathsf{GETN}\ (\lambda\mathsf{x}.\mathsf{C}))\ x\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu_0[\mathsf{x} \mapsto x]\ s\ h$ | |
| $(\mathsf{JMPK}\ \mathsf{y}\ \mathsf{x}\ \mathsf{t})\ \mu\ s\ h$ | $\Rightarrow\ h(\mu(\mathsf{y}))\ \mu(\mathsf{x})\ \mu(\mathsf{t})\ s\ h$ | |
| $(\mathsf{GETNT}\ (\lambda\mathsf{x}.\lambda\mathsf{t}.\mathsf{C})\ x\ t\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu_0[\mathsf{x} \mapsto x, \mathsf{t} \mapsto t]\ s\ h$ | |
| $(\mathsf{CASE}\ \mathsf{t}\ \overline{t_i \rightarrow \langle l_i, \mathsf{x}_0\, \overline{\mathsf{x}}_i \rangle})\ \mu\ s\ h$ | $\Rightarrow\ h(l_j)\ \langle l_j, \mu(\mathsf{x}_0)\, \overline{\mu(\mathsf{x})}_i \rangle\ s\ h$ | if $\exists j.\mu(\mathsf{t}) = t_j$ |
| | $\Rightarrow\ h(l_d)\ \langle l_d, \mu(\mathsf{x}_0)\, \overline{\mu(\mathsf{x})}_d \rangle\ s\ h$ | otherwise[2] |
| $(\mathsf{GETA}\ (\lambda\langle l, \mathsf{x}_0\, \overline{\mathsf{x}}\rangle.\mathsf{C})\ \langle l, x_0\, \overline{x}\rangle\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu_0[\mathsf{x}_0 \mapsto x_0, \overline{\mathsf{x} \mapsto x}]\ s\ h$ | |
| $(\mathsf{GETARG}\ (\lambda\overline{\mathsf{x}}_n.\mathsf{C}))\ \mu\ (\overline{x}_n\ s)\ h$ | $\Rightarrow\ \mathsf{C}\ \mu[\overline{\mathsf{x} \mapsto x}]\ s\ h$ | |
| $(\mathsf{PUTARG}\ \overline{\mathsf{x}}\ \mathsf{C})\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu\ (\overline{\mu(\mathsf{x})}\ s)\ h$ | |
| $(\mathsf{ARGCK}\ n\ \mathsf{C}_{mp}\ \mathsf{C})\ \mu\ (\overline{x}_m\ \langle l, \rho\rangle\ s)\ h$ | $\Rightarrow\ \mathsf{C}\ \mu\ (\overline{x}_m\ \langle l, \rho\rangle\ s)\ h$ | if $m \geq n$ |
| | $\Rightarrow\ \mathsf{C}_{mp}\ \mu\ (\overline{x}_m\ \langle l, \rho\rangle\ s)\ h$ | otherwise |
| $(\mathsf{GETK}\ (\lambda\langle l, \overline{\mathsf{x}}\rangle.\mathsf{C}))\ \mu\ (\langle l, \overline{x}\rangle\ s)\ h$ | $\Rightarrow\ \mathsf{C}\ \mu[\overline{\mathsf{x} \mapsto x}]\ s\ h$ | |
| $(\mathsf{PUTK}\ \langle l, \overline{\mathsf{x}}\rangle\ \mathsf{C})\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu\ (\langle l, \overline{\mu(\mathsf{x})}\rangle\ s)\ h$ | |
| $(\mathsf{REFK}\ (\lambda\mathsf{x}.\mathsf{C}))\ \mu\ (\langle l, \rho\rangle\ s)\ h$ | $\Rightarrow\ \mathsf{C}\ \mu[\mathsf{x} \mapsto l]\ (\langle l, \rho\rangle\ s)\ h$ | |
| $(\mathsf{SAVE}\ (\lambda\mathsf{s}.\mathsf{C}))\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu[\mathsf{s} \mapsto s]\ s\ h$ | |
| $(\mathsf{DUMP}\ \mathsf{C})\ \mu\ (\langle l, \rho\rangle\ s)\ h$ | $\Rightarrow\ \mathsf{C}\ \mu\ \langle l, \rho\rangle\ h$ | |
| $(\mathsf{RESTORE}\ \mathsf{s}\ \mathsf{C})\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu\ \mu(\mathsf{s})\ h$ | |
| $(\mathsf{GETC}\ \mathsf{x}\ (\lambda\langle l, \overline{\mathsf{w}}\rangle.\mathsf{C}))\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu[\overline{\mathsf{w} \mapsto w}]\ s\ h$ | $\theta(\mathsf{x}, l, \overline{w})$ |
| $(\mathsf{PUTC}\ \overline{\mathsf{x} \overset{\pi}{=} \langle l, \overline{\mathsf{w}}\rangle}\ \mathsf{C})\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu'\ s\ h[\overline{x^* \mapsto \langle l, \overline{\mu'(\mathsf{w})}\rangle}]$ | $\mu' = \mu[\overline{\mathsf{x} \mapsto x^*}]^3$ |
| $(\mathsf{UPDC}\ \mathsf{x}\ \langle l, \overline{\mathsf{w}}\rangle\ \mathsf{C})\ \mu\ s\ h$ | $\Rightarrow\ \mathsf{C}\ \mu\ s\ h[\mu(\mathsf{x}) \mapsto \langle l, \overline{\mu(\mathsf{w})}\rangle]$ | |
| $(\mathsf{PUTP}\ \overline{\langle l_i, \mathsf{x}_0\rangle}_n\ (\lambda\mathsf{y}.\mathsf{C}))\ \mu\ (\overline{x}_m\ \langle l', \rho\rangle\ s)\ h$ | | $n > m$ |
| $\Rightarrow\ \mathsf{C}\ \mu[\mathsf{y} \mapsto y^*]\ (\langle l', \rho\rangle\ s)\ h[y^* \mapsto \langle l_{m+1}, \mu(\mathsf{x}_0)\, \overline{x}_m\rangle]$ | | |

1. $\theta(\mathsf{x}, l, \rho)$ iff $h(\mu(\mathsf{x})) = \langle l, \rho\rangle$ where $\mu$ and $h$ are given on each context
2. default $\rightarrow \langle l_d, \mathsf{x}_0\, \overline{\mathsf{x}}_d\rangle$ is selected
3. $x^*$ means $x$ is a fresh heap address

**Fig. 2.** Reduction Rules of L-machine

An example of reduction is shown below:

$$
\begin{aligned}
& (\mathsf{GETN}\ (\lambda\mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_s, \rangle.\mathsf{ARGCK}\ 3\ \mathsf{C}_{mp}\ \mathsf{C}_r)))\ z\ (f\ g\ x\ s_0)\ h \\
\Rightarrow\ & (\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_s, \rangle.\mathsf{ARGCK}\ 3\ \mathsf{C}_{mp}\ \mathsf{C}_r))\ \mu_0[\mathsf{z} \mapsto z]\ (f\ g\ x\ s_0)\ h \\
& \text{where } h(z) = \langle l_s, \rangle \\
\Rightarrow\ & (\mathsf{ARGCK}\ 3\ \mathsf{C}_{mp}\ \mathsf{C}_r)\ \mu_0[\mathsf{z} \mapsto z]\ (f\ g\ x\ s_0)\ h \\
& \text{where } \mathsf{C}_r \equiv \mathsf{GETARG}\ (\lambda\mathsf{f}\ \mathsf{g}\ \mathsf{x}.(\mathbf{let}\ l_a = \mathsf{C}_a \mathbf{in}\ \mathsf{C}_1)) \\
\Rightarrow\ & (\mathsf{GETARG}\ (\lambda\mathsf{f}\ \mathsf{g}\ \mathsf{x}.(\mathbf{let}\ l_a = \mathsf{C}_a \mathbf{in}\ \mathsf{C}_1)))\ [\mathsf{z} \mapsto z]\ (f\ g\ x\ s_0)\ h \\
\Rightarrow\ & (\mathbf{let}\ l_a = \mathsf{C}_a \mathbf{in}\ \mathsf{C}_1)\ \mu_0[\mathsf{z} \mapsto z, \mathsf{f} \mapsto f, \mathsf{g} \mapsto g, \mathsf{x} \mapsto x]\ s_0\ h \\
& \text{where } \mathsf{C}_1 \equiv \mathsf{PUTC}\ \mathsf{a} \overset{u}{=} \langle l_a, \mathsf{g}\,\mathsf{x}\rangle\ (\mathsf{PUTARG}\ \mathsf{x}\ \mathsf{a}\ \mathsf{C}_2),\ \mathsf{C}_2 \equiv \mathsf{JMPC}\ \mathsf{f} \\
\Rightarrow\ & (\mathsf{PUTC}\ \mathsf{a} \overset{u}{=} \langle l_a, \mathsf{g}\,\mathsf{x}\rangle\ (\mathsf{PUTARG}\ \mathsf{x}\ \mathsf{a}\ \mathsf{C}_2))\ \mu_0[\mathsf{z} \mapsto z, ..., \mathsf{x} \mapsto x]\ s_0\ h[l_a \mapsto \mathsf{C}_a] \\
\Rightarrow\ & (\mathsf{PUTARG}\ \mathsf{x}\ \mathsf{a}\ \mathsf{C}_2)\ \mu_0[\mathsf{z} \mapsto z, ..., \mathsf{x} \mapsto x, \mathsf{a} \mapsto a]\ s_0\ h[l_a \mapsto \mathsf{C}_a, a \mapsto \langle l_a, g\, x\rangle] \\
& \text{where } a \text{ is a fresh heap address} \\
\Rightarrow\ & (\mathsf{JMPC}\ \mathsf{f})\ \mu_0[\mathsf{z} \mapsto z, ..., \mathsf{x} \mapsto x, \mathsf{a} \mapsto a]\ (x\ a\ s_0)\ h[l_a \mapsto \mathsf{C}_a, a \mapsto \langle l_a, g\, x\rangle]
\end{aligned}
$$

Eventually, the L-machine will produce an answer $(x, h)$, if it exists, in which $x$ is an address of some value and $h$ is a final status of the machine's heap.

### 2.4 Compiling STG Programs

In order to compile STG programs, some auxiliary syntactic elements are needed: the partial applications $(pap\ w_0\ \overline{w})$, the indirection $(ind\ w)$, the standard constructors $(c\ \overline{w})$, selection continuations $(sel\ \lambda z.alts)$, the update continuation $(upd\ w_1\ w_2)$, and the halt continuation $(halt)$.

Our STG compiler is presented in Figure 3. Notice that a set of free variables in some syntactic element is annotated as a prefix of the element, such as $\{\overline{w}\}.\lambda\overline{x}.e$, for convenience; $\{\overline{w}_n\}$ is considered as $\{w_1, ..., w_n\}$. Such annotation is assumed to be calculated on the fly whenever needed. Notice also that symbol tables $\tau$ are used in compilation rules. A symbol table $\tau$ is a mapping of variables into registers. Similarly, $\tau_0$ is an empty symbol table.

$\tau ::= \tau_0 \mid \tau[x \mapsto \mathsf{x}]$ symbol table

As shown in Figure 3, our compiler is composed of five kinds of compilation rules: $B$ for bound expressions, $E$ for expressions, $A$ for alternatives, $C$ for continuations, and $X$ for some auxiliary expressions. The $P$ rule receives an STG program and produces an L-code program in the form of nested **let** expressions that would be flattened by hoisting transformation.

Note that, according to the compilation rules, all **let**-bound $\mathsf{C}$s are closed and may be hoisted out to the top level without difficulty. The hoisted version will form a single-level **let** expression, in which any **let**-bound $\mathsf{C}$ contains no other **let** expression. An example of compiling an STG program is shown below:

$$
\begin{array}{l}
\quad \mathbf{let}\ l_{upd} = \ ...\ \mathbf{in} \\
\quad \mathbf{let}\ l_s = B[\![\{\}.\lambda f\ g\ x.\ \text{let}\ a \overset{u}{=} g\ x\ \text{in}\ f\ x\ a]\!]\ n\ l_s\ \mathbf{in}\ ... \\
= \mathbf{let}\ l_{upd} = \ ...\ \mathbf{in} \\
\quad \mathbf{let}\ l_s = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_s, \rangle.\mathsf{ARGCK}\ 3\ (...)(\mathsf{GETARG}\ (\lambda \mathsf{f}\ \mathsf{g}\ \mathsf{x}. \\
\qquad \mathbf{let}\ l_a = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_a, \mathsf{g}\,\mathsf{x}\rangle.\mathsf{SAVE}\ (\lambda \mathsf{s}.\mathsf{PUTK}\ \langle l_{upd}, \mathsf{z}\,\mathsf{s}\rangle \\
\qquad\qquad (\mathsf{DUMP}\ (\mathsf{PUTARG}\ \mathsf{x}\ (\mathsf{JMPC}\ \mathsf{g})))))) \\
\qquad \mathbf{in}\ \mathsf{PUTC}\ \mathsf{a} \overset{u}{=} \langle l_a, \mathsf{g}\,\mathsf{x}\rangle\ (\mathsf{PUTARG}\ \mathsf{x}\ \mathsf{a}\ (\mathsf{JMPC}\ \mathsf{f})))))) \\
\quad \mathbf{in}\ ...
\end{array}
$$

From the above example, we get its hosisted version as follows:

$$
\begin{array}{lll}
\mathbf{let}\ l_{upd} & = & ... \\
\quad l_s & = & \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_s, \rangle\rangle.\mathsf{ARGCK}\ 3\ (...)(\mathsf{GETARG}\ (\lambda \mathsf{f}\ \mathsf{g}\ \mathsf{x}. \\
 & & \quad \mathsf{PUTC}\ \mathsf{a} \overset{u}{=} \langle l_a, \mathsf{g}\,\mathsf{x}\rangle\ (\mathsf{PUTARG}\ \mathsf{x}\ \mathsf{a}\ (\mathsf{JMPC}\ \mathsf{f})))))) \\
\quad l_a & = & \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_a, \mathsf{g}\,\mathsf{x}\rangle.(\mathsf{SAVE}\ (\lambda \mathsf{s}.\mathsf{PUTK}\ \langle l_{upd}, \mathsf{z}\,\mathsf{s}\rangle \\
 & & \quad (\mathsf{DUMP}\ (\mathsf{PUTARG}\ \mathsf{x}\ (\mathsf{JMPC}\ \mathsf{g}))))))) \\
\mathbf{in}\ ...
\end{array}
$$

Let $\mathsf{C}$ be obtained from compiling an STG program. It will be reduced by applying the reduction rule $\Rightarrow$, starting from $\mathsf{C}\ \mu_0\ s_0\ h_0$.

$$
\begin{array}{ll}
B[\![\{\overline{w}\}.\lambda\overline{x}_n.e]\!]\ \pi\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l,\overline{\mathsf{w}}\rangle.\mathsf{ARGCK}\ n \\
& \qquad (\mathsf{PUTP}\ \overline{\langle l_{pap,i},\mathsf{z}\rangle}_n\ (\lambda \mathsf{p}.\mathsf{REFK}\ (\lambda \mathsf{y}.\mathsf{JMPK}\ \mathsf{y}\ \mathsf{p}\ \lambda))) \\
& \qquad (\mathsf{GETARG}\ (\lambda\overline{\mathsf{x}}.E[\![e]\!]\ \tau_0[\overline{w\mapsto \mathsf{w}},\overline{x\mapsto \mathsf{x}}])))) \\
B[\![\{\overline{w}\}.c\ \overline{w}]\!]\ \pi\ l & = X[\![\{\overline{w}\}.c\ \overline{w}]\!]\ l \\
B[\![\{\overline{w}\}.e]\!]\ u\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l,\overline{\mathsf{w}}\rangle.\mathsf{SAVE}\ (\lambda \mathsf{s}.\mathsf{PUTK}\ \langle l_{upd},\mathsf{z}\,\mathsf{s}\rangle \\
& \qquad (\mathsf{DUMP}\ (E[\![e]\!]\ \tau_0[\overline{w\mapsto \mathsf{w}}]))))) \\
B[\![\{\overline{w}\}.e]\!]\ n\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l,\overline{\mathsf{w}}\rangle.E[\![e]\!]\ \tau_0[\overline{w\mapsto \mathsf{w}}]))
\end{array}
$$

$$
\begin{array}{ll}
E[\![x_0\ \overline{x}]\!]\ \tau & = \mathsf{PUTARG}\ \overline{\tau(x)}\ (\mathsf{JMPC}\ \tau(x_0)) \\
E[\![\mathrm{case}\ e\ \mathrm{of}\ \{\overline{w}\}.\lambda x.alts]\!]\ \tau & = \mathbf{let}\ l_{sel} = C[\![\{\overline{w}\}.sel\ \lambda x.alts]\!]\ l_{sel} \\
& \quad \mathbf{in}\ \mathsf{PUTK}\ \langle l_{sel},\overline{\tau(w)}\rangle\ (E[\![e]\!]\ \tau) \\
& \quad \text{where } l_{sel} \text{ fresh} \\
E[\![\mathrm{let}\ \overline{x_i \overset{\pi_i}{=} \{\overline{w}_i\}.b_i}\ \mathrm{in}\ e]\!]\ \tau & = \mathbf{let}\ \overline{l_i = B[\![\{\overline{w}_i\}.b_i]\!]\ \pi_i\ l_i} \quad (b_i \not\equiv c\ \overline{y}) \\
& \quad \mathbf{in}\ \mathsf{PUTC}\ \overline{\mathsf{x}_i \overset{\pi_i}{=} \langle l_i,\overline{\mathsf{w}}_i\rangle}\ (E[\![e]\!]\ \tau[\overline{x_i\mapsto \mathsf{x}_i}]) \\
& \quad \text{where} \quad l_i \equiv l_c \ \text{ and } \mathsf{w}_i \equiv \tau[\overline{x_i\mapsto \mathsf{x}_i}](y) \quad \text{if } b_i \equiv c\ \overline{y} \\
& \qquad\qquad\ l_i \text{ fresh and } \mathsf{w}_i \equiv \tau[\overline{x_i\mapsto \mathsf{x}_i}](w) \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{ll}
A[\![\{\overline{w}\}.c\ \overline{x}\rightarrow e]\!]\ l & = \mathsf{GETA}\ (\lambda\langle l,\mathsf{z}\overline{\mathsf{w}}\rangle.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_c,\overline{\mathsf{x}}\rangle.E[\![e]\!]\ \tau_0[\overline{w\mapsto \mathsf{w}},\overline{x\mapsto \mathsf{x}}])) \\
A[\![\{\overline{w}\}.\mathrm{default}\rightarrow e]\!]\ l & = \mathsf{GETA}\ (\lambda\langle l,\mathsf{z}\overline{\mathsf{w}}\rangle.E[\![e]\!]\ \tau_0[\overline{w\mapsto \mathsf{w}}])
\end{array}
$$

$$
\begin{array}{ll}
C[\![\{w_1,w_2\}.upd\ w_1\ w_2]\!]\ l & = X[\![\{w_1,w_2\}.upd\ w_1\ w_2]\!]\ l \\
C[\![\{\overline{w}\}.sel\ \lambda w_0.\overline{alt_i}]\!]\ l & = \mathbf{let}\ \overline{l_i = A[\![alt_i]\!]\ l_i} \\
& \quad \mathbf{in}\ \mathsf{GETNT}\ (\lambda \mathsf{w}_0.\lambda \mathsf{t}.\mathsf{GETK}\ (\lambda\langle l,\overline{\mathsf{w}}\rangle.\mathsf{CASE}\ \mathsf{t}\ \overline{t_i\rightarrow\langle l_i,\mathsf{w}_0\ \overline{\mathsf{w}_i}\rangle})) \\
& \quad \text{where} \quad \overline{l_i} \text{ fresh},\ alt_i \equiv \{\overline{w}_i\}.t_i\ldots\rightarrow\ldots \\
C[\![\{\}.halt]\!]\ l & = X[\![\{\}.halt]\!]\ l
\end{array}
$$

$$
\begin{array}{ll}
X[\![\{w_0,\overline{w}\}.pap\ w_0\ \overline{w}]\!]\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l,\mathsf{w}_0\,\overline{\mathsf{w}}\rangle.\mathsf{PUTARG}\ \overline{\mathsf{w}}\ (\mathsf{JMPC}\ \mathsf{w}_0))) \\
X[\![\{w\}.ind\ w]\!]\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l,\mathsf{w}\rangle.\mathsf{JMPC}\ \mathsf{w})) \\
X[\![\{\overline{w}\}.c\ \overline{w}]\!]\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{REFK}\ (\lambda \mathsf{x}.\mathsf{JMPK}\ \mathsf{x}\ \mathsf{z}\ c)) \\
X[\![\{w_1,w_2\}.upd\ w_1\ w_2]\!]\ l & = \mathsf{GETNT}\ (\lambda \mathsf{z}.\lambda \mathsf{t}.\mathsf{GETK}\ (\lambda\langle l,\mathsf{w}_1\,\mathsf{w}_2\rangle. \\
& \qquad \mathsf{RESTORE}\ \mathsf{w}_2\ (\mathsf{UPDC}\ \mathsf{w}_1\ \langle l_{ind},\mathsf{z}\rangle\ (\mathsf{JMPC}\ \mathsf{z})))) \\
X[\![\{\}.halt]\!]\ l & = \mathsf{GETNT}\ (\lambda \mathsf{z}.\lambda \mathsf{t}.\mathsf{GETK}\ (\lambda\langle l,\rangle.\mathsf{STOP}\ \mathsf{z}))
\end{array}
$$

$$
\begin{array}{ll}
P[\![\overline{decl},e]\!] & = \mathbf{let}\ l_{pap,m} = X[\![\{\overline{w}_m\}.pap\ \overline{w}_m]\!]\ l_{pap,m} \quad (m \geq 1) \\
& \qquad\ \ l_{ind} \quad = X[\![\{w\}.ind\ w]\!]\ l_{ind} \\
& \qquad\ \ l_{upd} \quad = X[\![\{w_1,w_2\}.upd\ w_1\ w_2]\!]\ l_{upd} \\
& \qquad\ \ l_{halt} \quad = X[\![\{\}.halt]\!]\ l_{halt} \\
& \qquad\ \ l_c \qquad = X[\![\{\overline{w}_n\}.c\ \overline{w}_n]\!]\ l_c \quad (\mathrm{T}\ \overline{c\ n} \in \overline{decl}) \\
& \quad \mathbf{in}\ \mathsf{PUTK}\ \langle l_{halt},\rangle\ (E[\![e]\!]\ \tau_0)
\end{array}
$$

**Fig. 3.** Compilation Rules for the STG Language

## 3 Representation

From now on, we will concentrate on the development of an L-code-to-Java compiler. In this section, the method with which the elements in the L-machine

is represented in Java language is explained. In the next section, the compiler is defined, based on the representations.

### 3.1 Closure

The class *Clo* is a base class that captures common elements among closures. It has a public member variable *ind* that is initialized to itself, and an abstract member function *code*().

```
public abstract class Clo {
    public Clo ind = this;
    public abstract Clo code();
}
```

Every closure, e.g. of label $l$, is represented by a class that extends the base class by adding some member variables as its free variables and overriding the *code*() of the base class as follows:

```
public class C_l extends Clo {
    public Object f_1, ... , f_n;
    public Clo code() { ... }
}
```

Note that member variables are declared as type *Object*, which is the superclass of all classes in Java. Whenever items specific to some inherited class, such as accessing member variables in the class, are needed, the *Object* class can be cast into an appropriate class in our compilation scheme.

### 3.2 Runtime System

Our runtime system is a class $G$ that equips static variables *node*, *tag*, *loopflag*, *sp*, *bp*, and *stk*.

```
public class G {
    public static Object node;
    public static int tag;
    public static boolean loopflag;

    public static int sp, bp;
    public static Object[] stk;
        ...
}
```

The node variable *G.node* and the tag variable *G.tag* are used for passing the address $x$ of a heap-allocated closure and the tag $t$ respectively, as shown in the reduction rule:

$$(\mathsf{GETNT}\ (\lambda \mathsf{x}.\lambda \mathsf{t}.\mathsf{C}))\ x\ t\ s\ h \Rightarrow \mathsf{C}\ \mu_0[\mathsf{x} \mapsto x, \mathsf{t} \mapsto t]\ s\ h$$

A register file does not require any gadgets. Instead, the mapping by the register file is simply resolved by the scope rule in Java languages. A stack is represented by a big array *G.stk* of type *Object*, and *G.sp* and *G.bp* point to the top and bottom elements of the top portion of the stack respectively. A heap is represented by the internal heap provided by the JVM.

Note that the stack represented by an array has to be carefully handled. First, after an object is deleted from the top of a stack, the deleted entry of the object must be filled with null in order to inform the JVM that the object in the entry may be unnecessary. Second, an array of type *Object* can contain only values of any classes (or reference data type [3]), but the insertion of primitive values of types such as int and boolean will cause type violation at compile-time. This prevents the passing of unboxed arguments that are represented by primitive values. Instead, we pass them through global registers which are represented by static variables of appropriate primitive types.

### 3.3 Updating

In conventional machines, updating is achieved by directly overwriting a value at a given address. However, the JVM provides no way to construct one object directly on top of another. Therefore, updates must be managed in an alternative way. To do this, a special class *Ind* is introduced.

```
public class Ind extends Clo {
    public Clo code() { return this.ind; }
}
```

Whenever any non-Whnf is allocated, an object of *Ind* class is created together and its *ind* field is set to point to the non-Whnf. Later, at the time the non-Whnf is updated with a value, the *ind* field is given the value. Refer to the compilation rule for PUTC and UPDC in Section 4.

Note that every object representing a thunk is reachable only through an additional indirection object of the class *Ind* in this implementation. This indirection cannot be easily removed if a garbage collector does not treat it specially as in the conventional implementations. Although it is not possible to build up a chain of indirections in this implementation, even one-lengthed chains may cause a space leak. Probably, all the approaches in compiling lazy functional languages for the JVM are expected to suffer from this problem since it is not possible to interact with the built-in garbage collector of the JVM in general.

### 3.4 Tail Call

The STGM mostly transfers control in tail calls. Every tail call can be implemented by a single jump instruction on conventional machines. Since no tail call instruction is provided in the JVM, they must be simulated by a tiny interpreter as follows:

```
public static void loop (Clo c) { while(loopflag) c=c.ind.code(); }
```

Every sequence of L-code instructions ends with one of JMPK, JMPC, and CASE, according to our compilation rule shown in Figure 3. The three instructions are translated into Java statements that return an object to which control is transferred. After the return, control will reach the assignment statement in the body of the while loop of the above interpreter, and the interpreter will call the object's *code*() if *loopflag* is true. Here, the *loopflag* is a static boolean variable to make the STGM go or stop.

## 4 Compilation

Our compiler receives a hoisted version of L-code program, and generates Java classes using compilation rules for L-code instructions shown in Figure 4. Basically, our compilation scheme considers each closure as one class, which has member variables as the closure's free variables and has one method that simulates the closure's instructions.

Recall that a hoisted version of L-code program forms a single-level **let** expression containing **let** bindings $\overline{l = \mathsf{C}}$. By hoisting transformation, each C contains no inner **let** expression. Hence, a simple driver is needed to generate a class for each binding.

The driver may introduce member variables of the class from the relevant free variables obtained by analyzing the C. For convenience's sake, the STG-to-L-code compiler is assumed to annotate them to the binding like $l = \{\overline{w}\}.\mathsf{C}$, so the driver will use the annotation to make the member variables. The body of *code*() in the class will be filled with the Java statements generated by applying our L-code-to-Java compiler to the C in the binding.

Our compiler neatly maps registers used in L-code into some variables in Java codes. Each register will be accessed by its name, rather than by some offset as in conventional implementations. Accessing variables by names is much better, as it does not require costly array accesses.

In general, for bound variables $\overline{\mathsf{x}}$ in some L-code ... ($\lambda\overline{\mathsf{x}}$. ...), local variables $\overline{x}$ of type *Object* are declared in $J$ rules. Sometimes, it is not necessary to declare such local variables, for example, as in the rule for GETA. The reason is that the required local variables are already declared. According to the rule for CASE, Java codes generated from this instruction are combined with those generated from alternatives, though they are not in the same closure in terms of L-code. Since free variables that occur in the alternatives must be declared in the closure in which the CASE resides, the free variables in the alternative can be accessed from the preceding declaration by the scope rule in Java language without any additional declaration as in GETA.

For SAVE and RESTORE, the base pointer $G.bp$ is stored and restored instead of an entire stack, which is a well-known technique explained by Peyton Jones [7]. Note that, in the case of SAVE ($\lambda$s.C), $J$ declares a variable $s$ of type int as the bound variable s because it will hold the value of $G.bp$.

Due to space limitations, we omit discussion of some details of the L-code-to-Java compiler, including unboxed values and primitives. We also omit discussion

$$
\begin{array}{lcl}
J[\![\mathsf{JMPC\ x}]\!] & = & G.node = x;\ \text{return}\ (Clo)x \\
J[\![\mathsf{JMPK\ x\ y\ t}]\!] & = & G.node = y;\ G.tag = t;\ \text{return}\ (Clo)x \\
J[\![\mathsf{GETN}\ (\lambda \mathsf{x.C})]\!] & = & Object\ x = G.node;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{GETNT}\ (\lambda \mathsf{x}.\lambda \mathsf{t.C})]\!] & = & Object\ x = G.node;\ \text{int}\ t = G.tag;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{GETA}\ (\lambda\langle l, \mathsf{x}\,\overline{\mathsf{w}}\rangle.\mathsf{C})]\!] & = & J[\![\mathsf{C}]\!] \\
J[\![\mathsf{PUTARG}\ \overline{\mathsf{x}}_n\ \mathsf{C}]\!] & = & push\ x_n; \ldots\ push\ x_1;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{GETARG}\ (\lambda\overline{\mathsf{x}}_n.\mathsf{C})]\!] & = & Object\ x_1,\ldots,x_n;\ pop\ x_1; \ldots\ pop\ x_n;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{PUTK}\ \langle l, \overline{\mathsf{x}}_n\rangle\ \mathsf{C})]\!] & = & C_l\ o = \text{new}\ C_l();\ o.f_1 = x_1; \ldots\ o.f_n = x_n;\ push\ o;\ J[\![\mathsf{C}]\!] \\
 & & \text{where } o \text{ fresh} \\
J[\![\mathsf{GETK}\ \lambda\langle l, \overline{\mathsf{x}}_n\rangle.\mathsf{C})]\!] & = & Object\ o;\ pop\ o; \\
 & & Object\ x_1 = ((C_l)o).f_1,\ \ldots\ , x_n = ((C_l)o).f_n;\ J[\![\mathsf{C}]\!] \\
 & & \text{where } o \text{ fresh} \\
J[\![\mathsf{REFK}\ (\lambda \mathsf{x.C})]\!] & = & Object\ x = G.stk[G.sp];\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{SAVE}\ (\lambda \mathsf{s.C})]\!] & = & \text{int}\ s = G.bp;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{DUMP\ C}]\!] & = & G.bp = G.sp;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{RESTORE\ s\ C}]\!] & = & G.bp = s;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{PUTC}\ \overline{\mathsf{x}_i \overset{\pi_i}{=} \langle l_i, \overline{\mathsf{w}^i_j}_{m_i}\rangle}_n\ \mathsf{C}]\!] & = & \ldots\ alloc_i;\ \ldots\ assign^i_j; \ldots\ J[\![\mathsf{C}]\!] \\
 & & \text{where}\ alloc_i \equiv C_{l_i}\ x_i = \text{new}\ C_{l_i}() \qquad \text{if}\ \pi_i = n \\
 & & \qquad\qquad \equiv Ind\ x_i = \text{new}\ Ind(); \qquad \text{if}\ \pi_i = u \\
 & & \qquad\qquad\quad x_i.ind = \text{new}\ C_{l_i}() \\
 & & \qquad assign^i_j \equiv x_i.f_j = w^i_j \qquad \text{if}\ \pi_i = n \\
 & & \qquad\qquad \equiv ((C_{l_i})x_i.ind).f_j = w^i_j \qquad \text{if}\ \pi_i = u \\
J[\![\mathsf{GETC\ y}\ (\lambda\langle l, \overline{\mathsf{x}}_n\rangle.\mathsf{C})]\!] & = & Object\ x_1 = ((C_l)y).f_1; \ldots; x_n = ((C_l)y).f_n;\ J[\![\mathsf{C}]\!] \\
J[\![\mathsf{UPDC\ x}\ \langle l, \overline{\mathsf{w}}_n\rangle\ \mathsf{C}]\!] & = & C_l\ o = \text{new}\ C_l();\ o.f_1 = w_1;\ \ldots\ o.f_n = w_n; \\
 & & ((Clo)x).ind = o;\ J[\![\mathsf{C}]\!] \quad \text{where } o \text{ fresh} \\
J[\![\mathsf{PUTP}\ \overline{\langle l_i, \mathsf{y}\rangle}_n\rangle\ (\lambda \mathsf{x.C})]\!] & = & \text{switch}(G.sp - G.bp)\ \{ \\
 & & \quad \text{case } 0 : C_{l_1}\ x = \text{new}\ C_{l_1}();\ x.f_1 = y;\ \text{break}; \ldots \\
 & & \quad \text{case } n-1 : C_{l_n}\ x = \text{new}\ C_{l_n}();\ x.f_1 = y; \\
 & & \qquad\qquad pop\ x.f_2; \ldots\ pop\ x.f_n;\ \text{break}; \\
 & & \}\ J[\![\mathsf{C}]\!] \\
\multicolumn{3}{l}{J[\![\mathsf{CASE\ t}\ \overline{c_i \rightarrow \langle l_i, \mathsf{x}_0\,\overline{\mathsf{x}}_i\rangle}\ \text{default} \rightarrow \langle l_d, \mathsf{x}_0\,\overline{\mathsf{x}}_d\rangle]\!]} \\
 & = & \text{switch}(t)\ \{\ldots\ \text{case}\ c_i : J[\![\mathsf{C}_{l_i}]\!]\ \ldots\ \text{default} : J[\![\mathsf{C}_{l_d}]\!]\} \\
J[\![\mathsf{ARGCK}\ n\ \mathsf{C}_1\ \mathsf{C}_2)]\!] & = & \text{if}(n > G.sp - G.bp)\ \text{then}\ \{\ J[\![\mathsf{C}_1]\!]\ \}\ \text{else}\ \{\ J[\![\mathsf{C}_2]\!]\ \} \\
J[\![\mathsf{STOP\ y}]\!] & = & G.loopflag = \text{false};\ \text{return null}; \\
 & & \\
push\ X & = & G.sp{+}{+};\ G.stk[G.sp] = X \\
pop\ X & = & G.sp{-}{-};\ X = G.stk[G.sp+1];\ G.stk[G.sp+1] = \text{null}
\end{array}
$$

**Fig. 4.** Compilation Rules for L-code

of the known-call optimization to pass arguments via some agreed static variables rather than via the costly stack.

An example of compiling the L-code instructions bound to $l_a$ presented in Section 2.4 will be shown as follows. This example shows well the relationship that a **let** binding is directly translated into one Java class. Since our STG-

to-L-code compiler expresses each closure with a **let** binding, this example also shows the direct relationship between a closure and a class.

```
public class C_{l_a} extends Clo {          // l_a =
    public Object f1, f2;                   // {g, x}.
    public Clo code() {
        Object z = G.node;                  // GETN λz.
        Object g = ((C_{l_a})z).f1;         // GETC λ⟨l_a, g x⟩.
        Object x = ((C_{l_a})z).f2;
        int s = G.bp;                       // SAVE λs.
        C_{l_upd} o = new C_{l_upd}();      // PUTK ⟨l_upd, z s⟩
        o.f1 = z; o.f2 = s;
        G.sp ++; G.stk[G.sp] = o;
        G.bp = G.sp;                        // DUMP
        G.sp ++; G.stk[G.sp] = x;           // PUTARG x
        return (Clo)g;                      // JMPC g
    }
}
```

# 5 Benchmarking

We experiment with five small Haskell programs that have been used by Meehan and Joy [5]: fib 30, edigits 250, prime 500, soda, and queen 8. A SUN UltraSPARC-II workstation with a 296MHz processor, 768Mbytes of memory and Solaris version 2.5.1 is used. For compilers, GHC 4.04, Hugs98 Feb-2000, and Sun JIT compiler version 1.2.2 are used. To obtain optimized versions of STG programs, GHC are executed with `-O2` option.

## 5.1 STG-Level Optimizations

One of advantages in using the STGM is that many STG-level optimizations in GHC can be exploited freely. Ideally, our implementation can be imagined as a back end of GHC. In our prototype, our own STG-like functional language is defined, showing that the language is capable of precisely expressing the operational behaviors of the STG language in terms of the STGM. We extracted STG programs by running GHC with the option `-ddump-stg` from our Haskell benchmarks. Then the STG programs are semi-automatically modified according to the syntax of our own STG-like language. The main reason for the modification is that our implementation covers only a part of the libraries implemented in GHC. This translation is quite straightforward. Then, the modified STG programs are compiled into Java programs according to our compilation rules.

## 5.2 Results

Table 1 compares code sizes in bytes. For GHC, the sizes express the dynamically-linked executables stripped of redundant symbol table information. For Hugs, no

**Table 1.** Code Sizes in Bytes

| Pgms | GHC | JIT:unopt | JIT:opt |
|---|---|---|---|
| fib | 268,028 | 24,830 (037 classes) | 18,610 (034 classes) |
| edigits | 283,092 | 113,913 (135 classes) | 64,327 (092 classes) |
| prime | 280,248 | 74,161 (096 classes) | 50,025 (070 classes) |
| queen | 274,816 | 108,705 (134 classes) | 75,619 (101 classes) |
| soda | 302,972 | 335,873 (388 classes) | 185,496 (203 classes) |

sizes are given, as compiled codes exist only within the develpment environment. For our compiler, the sizes are obtained by summing up the sizes of class files produced from each program itself and the runtime system. The number in each parenthesis is the number of classes. Note that our runtime system consists of 4 classes with 2972 bytes.

Table 2 compares execution times in seconds. Each entry is the sum of user time and system time. The execution time is measured by UNIX `time` command; each benchmark is run five times and the shortest execution time is chosen. For Hugs, each execution time is calculated by a timer supported in Hugs and it excludes the time on parsing, type checking, and G-code generation.

### 5.3 Discussion

According to Table 1, each program consists of relatively many classes, since one class is generated for each closure. Particularly, the size of the unoptimized soda program is larger than that of the binary soda program generated by GHC, which stems from generating more updatable expressions by the naive translation of the string lists. As Wakeling did in his JFP paper [11], we can similarly reduce the bytecode size of generated classes by merging classes that have the same types of fields. By modifying only the L-code-to-Java compiler, we can cut the bytecode size by half and the number of classes is reduced to about 20 in all benchmarks.

According to Table 2, the execution times of generated Java programs lie between those of the relevant programs by GHC and Hugs. First, it is not surprising that the generated Java programs are slower than binary executables by GHC. The JVM is known to make memory allocation more costly, it ignores strong typedness of the Haskell programs so that it repeats unnecessary runtime checks, every tail call must be simulated by a return and a call instead of a single jump instruction, and the simulated stack incurs unnecessary cost for checking array out-of-bound index and assigning null after a pop.

Second, the generated Java programs tend to outperform the execution times in Hugs, particularly when GHC's STG-level optimizations are applied; in that case, all programs except soda run faster. In the case of soda, because the warming-up time of the JVM varies from 0.68 to 0.74 seconds, any generated Java program for soda cannot run faster than in Hugs. By refining the way of interaction between a primitive and a case expression in the STG-to-L-code

**Table 2.** Execution Time in Seconds

| Programs | GHC | Hugs | JIT:unopt | JIT:opt |
|---|---|---|---|---|
| fib | 0.18s | 106.48s | 25.70s | 5.72s |
| edigits | 0.16s | 3.10s | 9.15s | 2.42s |
| prime | 0.14s | 3.30s | 86.38s | 1.97s |
| queen | 0.07s | 5.79s | 5.35s | 2.29s |
| soda | 0.03s | 0.41s | 2.26s | 1.59s |

compiler, all the above execution times can be cut down to a half, on average. These results show that our systematic compiler also generates Java programs which run at a reasonable execution speed.

The STG-level optimizations by GHC dominantly affect the execution times of all programs. In the case of prime, they are particularly effective. Its explanation may be given in the aspect of total heap allocation since execution time is usually affected by the amount of heap allocation in lazy functional languages. An optimized version of prime allocates 35,061,856 bytes, while unoptimized version allocates 2,603,827,792 bytes. The unoptimized prime allocated 74.3 times more heap objects than in its optimized version. By this observation, we notice that more reduction in heap allocation tends to affect more reduction in execution time. This is a reasonable expectation as in conventional implementations, and we believe this is also true for the implementations on the JVM, particularly because the cost of memory allocation is known to be more expensive in the JVM than in the conventional machines.

## 6 Related Works

The previous works based on the STGM are Tullsen [9] and Vernet [10]. As done in the original paper regarding the STGM [7], both papers explained their own mapping informally based on examples. With these descriptions, we cannot reproduce their compilers. Furthermore, we cannot compare the performance, since no experimental result for performance is provided.

Other works are based on either G-machine or $\langle \nu, \mathrm{G} \rangle$-machine. Wakeling's compiler [11] has two distinct features. It employs a list of small local stacks instead of one single big stack, which comes from a feature of $\langle \nu, \mathrm{G} \rangle$-machine, and it also generates much smaller number of classes over quite big benchmarks by efficient representations. The generated Java programs run at competitive speed with Hugs. Wakeling explained that the size of class files affects performance. Although we didn't compare our compiler with his in terms of performance, our compiler based on the STGM is believed to be advantageous in that it can exploit STG-level optimizations as well as his idea on the efficient representations to reduce the size of class files. Further work is required.

Meehan and Joy's compiler [5] represents functions by methods in a single class and uses the Java reflection package to access the methods. Their generated Java programs by the compiler runs slower than with Hugs.

Recently, we heard that Erik Meijer, Nigel Perry, and Andy Gill are working on a Java back end for GHC, though no documentation is available for their work [6].

Some research has been done on compiling strict functional languages for the JVM. Benton, Kennedy, and Russel have taken with Standard ML(SML) as their source language, improving performance through extensive optimizations and providing SML with an interface to Java [1][2]. Their approach is similar to ours with Haskell.

## 7 Conclusion and Further Works

In this study, a systematic specification of the STGM is presented by defining our STG-to-L-code compiler and L-machine. It is a new attempt to describe the mapping accurately by the provision of *concrete* compilation rules. It is compact enough to be presented in a few pages, but exposes every aspect for the mapping of the STGM. Each instruction of L-code is directly mapped onto some Java statements, and each **let** binding identifies all the instructions within one closure so that the closure is nicely mapped onto one class in Java.

Based on the specification, a concrete compilation rule for the JVM is defined. In this phase, the goal of implementing the STGM is decomposed into the subgoals of implementing L-code instructions. We have defined an L-code-to-Java compiler with our representation decisions.

Since our approach is based on the STGM, the generated Java programs are indirectly amenable to the STG-level optimizations in GHC. The initial performance is measured with five small Haskell benchmarks. Combining the optimizations, promising results are obtained for the benchmarks.

As a further work, we hope to devise more ingenious Java representation; our representation scheme is a bit straightforward. Also, it is important to apply our compiler to bigger benchmarks, and this will give more decisive figures for performance of the translated Java programs. Finally, we hope to more fully exploit the features of the two-level translation; one may analyze flow information in terms of L-code to optimize L-code programs, and one may devise Java-related optimizations relevant to our compilation scheme. Each work could be done separately.

## Acknowledgment

We thank anonymous referees for their many helpful comments. The first author also thanks Hyeon-Ah Moon for her encouragement. This work is supported by Advanced Information Technology Research Center.

## References

1. N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *Proceedings of the 4th ACM SIGPLAN Conference on Functional Prgramming*, pages 126–137, 1999.

2. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java Bytecodes. In *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Prgramming*, pages 129–140, 1998.
3. M. Campione and K. Walrath. *The Java Tutorial (2nd Ed.)*. Addison Wesley, March 1998.
4. T. Lindholm and F. Yellin. *The Java$^{TM}$ Virtual Machine Specification (2nd Ed.)*. Addison Wesley, 1999.
5. G. Meehan and M. Joy. Compiling Lazy Functional Programs to Java Bytecode. *Software-Practice and Experience*, 29(7):617–645, June 1999.
6. S. L. Peyton Jones. A Java Back End for Glasgow Haskell Compiler. The Haskell Mailing List haskell@haskell.org (http://www.haskell.org/mailinglist.html), May 2000.
7. S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.
8. S. L. Peyton Jones and A. L. M. Santos. A Transformation-based Optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.
9. M. Tullsen. Compiling Haskell to Java. 690 Project, Yale University, September 1997.
10. A. Vernet. The Jaskell Project. A Diploma Project, Swiss Federal Institute of Technology, February 1998.
11. D. Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):579–603, November 1999.
12. M. Wand. Correctness of Procedure Representations in Higher-Order Assembly Language. In S. Brookes, editor, *Proceedings Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311. Springer Verlag, 1992.