A Practical Implementation of Root Optimization in G-machine

Kwanghoon Choi, Chiung O, and Taisook Han

September 1, 1998

Korea Advanced Institute of Science and Technology
Email: {khchoi, ocu, han}@cs.kaist.ac.kr

Abstract

In functional languages based on lazy evaluation, recursive call with static arguments can create thunks containing the same partial application repeatedly. M.P.Jones introduced the ROOT instruction in G-machine to get the partial application in thunks so that those overheads could be reduced. This is called root optimization. Although root optimization can reduce runtime space, it incurs time overhead to search a common portion of thunks. It also prevents from using the vag representation and tail call optimization. These cause difficulties in employing root optimization for practical compilers.

Our goal is to implement root optimization with feasible execution-time overhead, using the vag representation and tail call optimization. We propose a solution which transforms a function into one that can share the partial application of thunks safely. We introduce new tags and runtime system routines for the tags. We implement the idea on Chalmers Haskell compiler (HBC) to show its usefulness and suggest heuristic to apply root optimization only when the number of dynamic arguments is one or zero.

1 Introduction

Lazy evaluation is a way of passing unevaluated arguments to a function and sharing the resulting values afterward, once the arguments are evaluated in the function. The distinctions require making thunks over arguments and updating thunks with values for later uses.

A thunk is a block containing code and environment over a given expression. Once lazy evaluation is used, thunks should be created before every function call in general. It results in the runtime overhead of thunk creation. Moreover, when redundant arguments are involved in recursive calls, some related information may be stored in thunks redundantly.

Of all the traditional approaches to reduce the overhead of thunk creation, strictness analysis can pick out which thunks may be evaluated safely before function calls, and β-reduction (or inlining) can pick out which thunks may be deferred to be created or eliminated. These approaches, however, have some limitations. First, the implementation skills on strictness analysis seem to be immature in practice until now, and arguments can be inherently nonstrict. Second, β-reduction (or inlining) allows only a finite number of applications because it is not known how many recursive calls occur in compile-time.

Another approach tries to share redundant information between thunks to reduce the overhead of thunk creation[10][17][3]. Thunks contain redundant information during recursive call in the case the following conditions:
- A recursive function has one or more static arguments.
- All the recursive call instances in the function occur as an argument of some function.
- A thunk is created to represent unevaluated forms for every recursive call instance.

The approach uses the fact that the redundant partial application of thunks can be shared and only the rest need to be constructed newly during recursive call. Here is a map function satisfying above conditions, written in Haskell[6].

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (x:xs) &= f \; x : \text{map } f \; xs
\end{align*}
\]

User-defined functions(e.g., map) may have static arguments. Extra parameters may be added by some compiler-driven transformation, such a lambda lifting[8] and overloading resolving[5] and they become static arguments.

How to share some redundant information of thunks depends on underlying abstract machines. There are two kinds of implementations: static argument transformation on STGM and root optimization on G-machine. Static argument transformation introduces a new let to contain the redundant information as a nonlocal environment, and a new internal function to behave same but differently in access to the redundant information[17][3]. A thorough experimental result on Glasgow Haskell compiler(GHC)\(^1\) is presented using nofib benchmarks[16]. Root optimization introduces a new G-machine instruction to get a partial application of the thunk on evaluation. The new instruction replaces original instructions which would construct the common partial application[10]. It is implemented on Hugs\(^2\), but little experimental result has been presented.

Root optimization has some limitations. First, the new instruction is time-consuming to get a partial application. Second, it is difficult to share when a thunk is in the vap representation, rather than in the ap representation. The vap representation is more efficient than the ap representation with respect to time and space. Third, tail call optimization makes difficult to get the common portion. The limitation hinders using root optimization in more realistic compiler like HBC\(^3\).

We suggest an alternative implementation method of root optimization using:

- New tags and specialized runtime routines to get the common portion more quickly
- Worker/wrapper transformation[13] to use both the vap representation and tail call optimization.

With our implementation method, root optimization can be used to reduce heap space with small execution-time overhead, and no more problem on interacting with the vap representation and tail call optimization. We implement our method on HBC to show its feasibility, and present experimental results.

The remainder of this paper is organized as follows. In Section 2, we describe G-machine and root optimization. Our revised root optimization is described in Section 3. The implementation and experimental results are outlined in Section 4. We discuss static argument transformation and related works Section 5. We conclude our paper in Section 6.

\(^{1}\)http://www.dcs.gla.ac.uk/~fp/software/ghc/

\(^{2}\)http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html

\(^{3}\)http://www.cs.chalmers.se/~augustss/hbc.html
2 Root optimization

2.1 G-machine

G-machine is a fast implementation of graph reduction based on compiling supercombinators. User-defined functions are compiled into instruction sequences for G-machine. The instructions construct and manipulate explicitly graphs for expressions to reduce the expressions to their values. G-machine compilation rules and state transition rules are shown in Figure 3 and Figure 4[7][9].

The graph for a function application, which is a thunk, has two representations: the ap representation and the vap representation. These are described in Figure 1. The ap representation always has two entries: one for function part and the other for argument part. The vap representation has the varying number of entries. In HBC, the vap representation is restricted to be used when the number of arguments is equal to the arity of a function. Hence, we have no need to store additionally how many entries are in the vap representation, because the arity is found in the function descriptor vector.

The vap representation is cheaper to build than the ap representation because it allocates fewer words on the heap. It is also advantageous to unwind operation. Therefore, the vap representation is preferable whenever possible.

Unwind selects the next redex to evaluate in normal order reduction, it rearranges the stack as shown in Figure 2, and it begins executing the code for the function. The stack will have the pointers directed to both arguments and the root of the current redex after unwind.

The steps for unwind are different according to the representation of a thunk. Initially, the top of the stack has a pointer to any redex (or a thunk), e.g. \( fe_1 e_2 e_3 \). Then,
\[ \mathcal{F}[x_1 \ldots x_k = e] = \mathcal{R}[e] [x_1 = k, \ldots, x_k = 1] \]

\[ \mathcal{R}\{\text{case } e \text{ of } \{ \ldots \mid c_i \cdot v_i \ldots v_{k_i} \Rightarrow e_i \mid \ldots \} \} r n = \]
\[ \mathcal{E}[e] r n; \text{PUSH } 0; \text{CASE } (t_1, l_1) \ldots (t_n, l_n); \]
\[ \ldots \]

LABEL \(l_i\); PUSH 0; SPLIT \(t_i k_i\);
\[ \mathcal{R}[e_i] r [v_{k_i} = n + 2, \ldots, v_1 = n + 1 + k_i] (n + 1 + k_i); \]
\[ \ldots \]

\[ \mathcal{R}[x] r n = \text{PUSH } (n - r(x)); \text{MOVE } n; \text{POP } (n - 1); \text{EVAL}; \]
\[ \text{UPDATE } 1; \text{UNWIND} \]
\[ \mathcal{R}[f] r n = \text{PUSHGLOBAL } f; \text{MOVE } n; \text{POP } (n - 1); \]
\[ \text{UPDATE } 1; \text{UNWIND} \]
\[ \mathcal{R}[i \, e_1 \ldots e_m] r n = \mathcal{S}[i \, e_1 \ldots e_m] r n; \text{JFUN } m \]
\[ \mathcal{R}[e] r n = \mathcal{E}[e] r n; \text{UPDATE } (n + 1); \text{POP } n; \text{UNWIND} \]

\[ \mathcal{C}[f] r n = \text{PUSHGLOBAL } f \]
\[ \mathcal{C}[x] r n = \text{PUSH } (n - r(x)) \]
\[ \mathcal{C}[f_1 \ldots e_m] r n = \mathcal{C}[e_m] r n; \ldots; \mathcal{C}[e_1] r (n + m - 1); \]
\[ \text{MKVAP } f \text{ (arity } f); \text{MKAP} \ldots \text{MKAP}, \text{ if arity } f \leq m \]
\[ \text{(m - arity } f) \]
\[ \mathcal{C}[f_1 \ldots e_m] r n = \mathcal{C}[e_m] r n; \ldots; \mathcal{C}[e_1] r (n + m - 1); \]
\[ \mathcal{C}[f] r (n + m); \text{MKAP} \ldots \text{MKAP}, \text{ otherwise} \]
\[ \mathcal{C}[\text{constructor } e_1 \ldots e_m] r n = \mathcal{C}[e_m] r n; \ldots; \mathcal{C}[e_1] r (n + m - 1); \]
\[ \text{CONSTR constructor } m \]

\[ \mathcal{S}[i \, e_1 \ldots e_m] r n \text{ rearranges the stack for a tail call.} \]
\[ \mathcal{E}[e] r n \text{ gives code that computes the value of } e \text{ and leaves a pointer to the result on top of the stack.} \]
\[ \mathcal{B}[e] r n \text{ gives code that computes the basic value of } e \text{ and leaves the result on top of the basic value stack.} \]

**Figure 3:** The compilation scheme for G-machine
1. \( \langle o, \text{EVAL} : c, n : s, G[n = AP n_1 n_2] : E, D \rangle \Rightarrow \langle o, \text{UNWIND} (.), n : () : G, E, (c, s) : D \rangle \\
2. \langle o, \text{EVAL} : c, n : s, G[n = \text{CONSTR} k n_1 \cdots n_m] : E, D \rangle \Rightarrow \langle o, c, n : s, G, E, D \rangle \\
   \quad \text{similarly for } \text{FUN} \ f \\
3. \langle o, \text{UNWIND} (.), n : s, G[n = AP n_1 n_2] : E, D \rangle \Rightarrow \langle o, \text{UNWIND} (.), n_1 : n : s, G, E, D \rangle \\
4. \langle o, \text{UNWIND} (.), n : s, G[n = \text{VAP} n_f n_1 \cdots n_k] : E[n_f = \text{FUN}(c, k)] : D \rangle \\
   \Rightarrow \langle o, c, n_1 \cdots n_k : s, G, E, D \rangle \\
5. \langle o, \text{UNWIND} (.), n_0 : \cdots : n_k : s, G[n_0 = \text{FUN} \ f, n_i = AP n'_i n'_j] : E[f = (k, c)] : D \rangle \\
   \Rightarrow \langle o, c, n'_0 : \cdots : n'_k : s, G, E, D \rangle \\
6. \langle o, \text{UNWIND} (.), n_0 : \cdots : n_k : () : G[n_0 = \text{FUN} \ f, E[f = (a, c)] : (c', s') : D & k < a \rangle \\
   \Rightarrow \langle o, c', n_k : s', G, E, D \rangle \\
7. \langle o, \text{JFUN} \ m : c, n : s, G[n = \text{FUN} \ f] : E[f = (m, c')] : D \rangle \Rightarrow \langle o, c', s, G, E, D \rangle \\
8. \langle o, \text{JFUN} \ m : c, n : s, G, E, D \rangle \Rightarrow \langle o, \text{MKAP} \cdots \text{MKAP:UPDATE} 1 : \text{RET} (.), s, G, E, D \rangle \\
9. \langle o, \text{RET} : c, n : () : G[n = \text{CONSTR} k] : E, (c', s') : D \rangle \Rightarrow \langle o, c', n : s', G, E, D \rangle \\
10. \langle o, \text{RET} : c, n : s, G[n = AP n_1 n_2] : E, D \rangle \Rightarrow \langle o, \text{UNWIND} (.), n : s, G, E, D \rangle \\
11. \langle o, \text{PUSH} \ m : c, n_0 : \cdots : n_m : s, G, E, D \rangle \Rightarrow \langle o, c, n_0 : \cdots : n_m : s, G, E, D \rangle \\
12. \langle o, \text{PUSHGLOBAL} \ f : c, s, G, E, D \rangle \Rightarrow \langle o, c, n : s, G[n = \text{FUN} \ f] : E, D \rangle \\
13. \langle o, \text{MOVE} \ m : c, n_0 : \cdots : n_m : s, G, E, D \rangle \Rightarrow \langle o, c, n_1 : \cdots : n_{m-1} : n_0 : s, G, E, D \rangle \\
14. \langle o, \text{POP} \ m : c, n_0 : \cdots : n_m : s, G, E, D \rangle \Rightarrow \langle o, c, s, G, E, D \rangle \\
15. \langle o, \text{UPDATE} \ m : c, n_0 : \cdots : n_m : s, G[n_0 = N_0, n_m = N_m] : E, D \rangle \\
   \Rightarrow \langle o, c, n_1 : \cdots : n_m : s, G[n_0 = N_0] : E, D \rangle \\
16. \langle o, \text{MKAP} : c, n_1 : n_2 : s, G, E, D \rangle \Rightarrow \langle o, c, n' : s, G[n' = AP n_1 n_2] : E, D \rangle \\
17. \langle o, \text{MKVAP} \ f \ k : c, n_1 : \cdots : n_k : s, G, E, D \rangle \\
   \Rightarrow \langle o, c, n' : s, G[n' = \text{VAP} n_f n_1 \cdots n_k, n_f = \text{FUN}(c, k)] : E, D \rangle \\
18. \langle o, \text{CASE} \cdots (t_i, l_i) \cdots : \text{LABEL} \ l : c, n : s, G[n = \text{CONSTR} t_i n_1 \cdots n_k] : E, D \rangle \Rightarrow \langle o, c, s, G, E, D \rangle \\
19. \langle o, \text{CONSTR} \ t : k : c, n_1 : \cdots : n_k : s, G, E, D \rangle \Rightarrow \langle o, c, n' : s, G[n' = \text{CONSTR} t n_1 \cdots n_k] : E, D \rangle \\
20. \langle o, \text{SPLIT} \ t : k : c, n : s, G[n = \text{CONSTR} k n_1 \cdots n_m] : E, D \rangle \Rightarrow \langle o, c, n_1 : \cdots : n_m : s, G, E, D \rangle 

Figure 4: The state transition rules for G-machine instructions
• If the thunk is in the \textit{ap} representation, the pointers directed to \textit{ap} nodes are stored on stack following their left pointer until a \textit{fun} node is met. Then the number of pointers stored on stack is compared with the arity of the function to verify whether there are sufficient arguments. If so, the pointers, which will be an argument, are replaced with the right pointer of their corresponding \textit{ap} node. (Rules 3,5,6 in Figure 4)

• If the thunk is in the \textit{vap} representation, it is not necessary to follow some pointers and to check the number of arguments if the exact number of arguments are known. This is a strategy in HBC\cite{9}. The pointers in the \textit{vap} representation are stored on the stack(Rule 4 in Figure 4)

Note that the stack keeps a pointer to the root of the current redex for updating operation. Lazy evaluation requires sharing the value of an expression, instead of reevaluating the expression whenever its value is needed. The pointer is used to find the updating location.

2.2 Tail call optimization

Tail call optimization is a traditional method to replace a tail call by a jump instruction with some argument arrangement, which would be done by the call. Clearly, it is unnecessary to retain the stack frame of calling function since it will never be used after the call.

In G-machine all the entries, except one for the root of the current redex, are emptied from the current context of stack, and then the pointers directed to arguments of called function fills the entries instead. It is called dispatching\cite{12}

Note that the entry for the root of a redex is left untouched. It is legal and also efficient not to update the root of the redex with a graph representing the tail call application since the graph has not been a value. Sometimes, however, it makes more active cells during garbage collection, so it may cause a space leak. To prevent from the space leak, we may update the root of the current redex with a \textit{hole}\cite{11}. It is time-consuming but can prevent a possible space leak. It is also legal since it is necessary to keep only the location of a redex for sharing, not its content.

2.3 Original implementation and its limitations

Root optimization is a method to reuse a partial application of the current redex. It is applied to the case that some partial application of a recursive call instance is identical to the part of the current redex. It happens due to static arguments and thunk creation over the recursive call instance\cite{10}. The sharing of a partial application is performed by a new ROOT instruction.

\begin{figure}[h]
\centering
\begin{verbatim}
\langle o,ROOT l m : c,nn_0 : \cdots : nn_m : s,G[nn_m = n_k = AP n_{k-1}n'_k,\cdots,n_i = AP n_{i-1}n'_i,n_0 = FUN f],E[f = (k,c)],D \rangle \Rightarrow \langle o,c,n_{k-1} : nn_0 : \cdots : nn_m : s,G,E,D \rangle
\end{verbatim}
\caption{The Transition Rule for ROOT Instruction}
\end{figure}

The new ROOT instruction finds a partial application by descending \textit{l} times starting from the root of the current redex, and then it puts the partial application on the top of stack, as shown in Figure 5. The current redex is found in the \textit{(m+1)}-th entry from the top of stack.

A new compilation rule for root optimization is shown in Figure 6. The new C rule over a call instance generates instructions which create a thunk with some partial application of the
\[
C[f \ x_1 \ \cdots \ x_{m'} \ \mathit{e}_{m'+1} \ \cdots \ \mathit{e}_m] \ r \ n = C[\mathit{e}_m] \ r \ n; \cdots; C[\mathit{e}_{m'+1}] \ r \ (n + m - m' - 1); \\
\text{ROOT} (k-m')(n + m - m') \ ; \mathit{MKAP} \cdots \mathit{MKAP}, \\
\text{if it is a recursive call instance and } f = \lambda x_1 \cdots \lambda x_m. \mathit{e}
\]

for an arity of \( f, k \)

**Figure 6:** Compilation Scheme for Root Optimization

The current redex. It is easy to verify whether a partial application of the call instance is common with a redex.

The \textit{map} function has a recursive instance \textit{map \( f \ \mathit{xs} \)} as an argument of \((:\)\), and \( f \) is a static argument. We can apply root optimization to the \textit{map} function. If the second argument of \textit{map} is a non-empty list, e.g. \( \mathit{xs} \), then \( \mathit{f} \ \mathit{xs} : \textit{map} \ f \ \mathit{xs} \) is constructed by the definition of \textit{map}. According to the \( C \) rule shown in Figure 6, \textit{map} \( f \ \mathit{xs} \) is compiled into

\[
C[\textit{map} \ f \ \mathit{xs}] \ r \ n = \text{PUSH}(n - r(\mathit{xs})); \text{ROOT} \ 1 \ (n + 1); \mathit{MKAP}, \text{ for some } r, n
\]

An example of \textit{ROOT} instruction for \textit{map} is described in Figure 7. When the \textit{ROOT} instruction is executed, the \textit{ap} node representing \textit{map} \( f \) of the current redex is the same at every recursive call.

**Figure 7:** \textit{ROOT} 1 \((n + 1)\) for some \( n \)

This implementation of root optimization has several problems whenever we use it with the \textit{wap} representation or with tail call optimization. Although these are not implemented on Hugs, these has been believed to be useful in general[9].

- It is natural to use the \textit{ap} representation for sharing instead of the \textit{wap} representation, but it consumes more space and time in thunk creation and unwind operation. The \textit{wap} representation should be used freely if root optimization is not concerned.
- \textit{ROOT} instruction is time-consuming because it wastes linear time in the number of arguments for find a common partial application.
- The common portion of a thunk should be found as simple as possible to reduce execution-time. If tail call optimization is used, the root of redex may not contain the intended partial application. That’s why we need more complex \textit{ROOT} instruction.
3 Revised Root Optimization

Our intended stack configuration for the revised root optimization contains an additional entry for a common partial application shown in Figure 8. Then it is enough to access the entry to get the partial application.

How do we get the stack configuration without losing so much time? Our solution is to use new tags $ap_{k,l}$ for some $k$ and $l$, and to use worker/wrapper transformation[13] for an adjustment to the thunk representation.

3.1 $AP_{k,l}$

We introduce $ap_{k,l}$ node which is similar to $ap$ node. $Ap_{k,l}$ node means that an application starting from the node has $k$ arguments and that its common partial application is found by deleting last $l$ arguments. During unwind, when the $ap_{k,l}$ node is met, the intended stack configuration is made immediately by the following rule.

$$\langle o, UNWIND : (), n_k : s, G[n_k = AP_{k,l} n_{k-1} n'_1, n_i = AP n_{i-1} n'_i, n_0 = FUN f], E[f = (k, c)], D \rangle$$

gives $\langle o, c, n_{k-1} : n'_1 \cdots n'_k : n_k : s, G, E, D \rangle$

Figure 9: The transition rule for a new UNWIND instruction

Note that the ROOT instruction follows the left pointers of the $ap$ nodes to get some partial application. A group of the left pointers is called a spine. The unwind instruction also follows a spine to get a redex. Two instructions are very similar, so the common partial application is found during unwind, when $l$ is known. By this observation, we can get a common partial application more efficiently without using the ROOT instruction, which is expensive because it takes linear time in the number of arguments. In the $ap_{k,l}$ node, $l$ is needed to find the position of a common partial application.

Why do we need $k$ which is the number of arguments? Although the $ap_{k,l}$ node helps to find efficiently a common partial application, it is still inefficient because it needs an argument rearrangement as shown in Rule 5 in Figure 4. The argument rearrangement takes linear time in the number of arguments, too. However, the $wap$ representation does not need the argument rearrangement because all arguments are already guaranteed to be in the $wap$ node. If an $ap_{k,l}$ node is the root node of a redex and it guarantees the argument satisfaction, it is not necessary to rearrange arguments after following a spine to verify argument satisfaction. Argument rearrangement can be done together during the spine traversal of unwind operation.
\[ f_{x_1 \cdots x_k} = e, \quad \text{where static arguments are } x_1 \cdots x_{k-l} \]
\[ \Rightarrow \]
\[ f_{x_1 \cdots x_k} = (f_{w\cdot x_1 \cdots x_k})\{-\textsc{BUILD} \ k \ l\} \]
\[ f_{w\cdot x_1 \cdots x_k} = e[(f_{w\cdot e_1 \cdots e_m})\{-\textsc{ROOTOPT} \ k \ l\}/(f \ e_1 \cdots e_m)] \]
for all recursive call instances

**Figure 10**: Worker/wrapper transformation

As a result, we can reduce the number of traversing a spine from three times to one by using the \( ap_{k,l} \) node. It is the key to our implementation of root optimization without much execution time.

The new rule for unwind works well only if a thunk is constructed correctly with \( ap_{k,l} \) for some \( k \) and \( l \). In the next section, we will explain how to construct a thunk with \( ap_{k,l} \).

### 3.2 Worker and Wrapper

Assume we apply root optimization to a function \( f \). The body of \( f \) should be compiled to use the intended stack configuration, so a common partial application could be get from the appropriate stack entry. When an application \( f \ e_1 \cdots e_m \) is a tail call or the application is in the \( v\!a\!p \) representation, it is rather difficult to make an appropriate stack configuration.

- The \( S \) compilation rule for a tail call needs to consider which stack configuration is used, so we need some control flow information.

- The new unwind rule shown in Figure 9 is not for the \( v\!a\!p \) node. We need other efficient, but probably complex, rule for the \( v\!a\!p \) node.

To build correctly a thunk starting with \( ap_{k,l} \), we apply worker/wrapper transformation as shown in Figure 10 without any control flow information. \{- -\} is an annotation for its prefix expression and \([e_1/e_2]\) is a rather abused but clearly understood notation for substitution. The substitution results in the worker which has all recursive call instances annotated with \textsc{Rootopt}, but it should be equivalent to the original function. The name of a worker is a new one so that no other function can call the worker directly. Wrapper bridges the control flow between worker and other functions, and it is specially compiled by annotating its body with \textsc{Build}.

Wrapper guarantees the intended stack configuration. Therefore, we can apply root optimization without restriction over tail call optimization or over using the \( v\!a\!p \) representation. The compilation rules for worker and wrapper are shown in Figure 11, and a new transition rule for making \( ap_{k,l} \) node is explained in Figure 12.

Note that we can improve the \( R \) compilation rule in Figure 11. Although the rule instructs to make a new thunk containing all arguments with \( ap_{k,l} \), it is enough to make a common partial application and to put it on the appropriate entry of stack.
\[ F[x_1 \cdots x_k = e] = \mathcal{R}[e] \quad \text{if } f_w \text{ is a worker} \]
\[ \mathcal{R}(f_w e_1 \cdots e_k) \{ \text{-BUILD } k \ell - \} \quad r \; n = C[e_k] \quad r \quad n; \cdots; C[e] \quad r \quad n + k - 1; C[f_w] \quad r \quad n + k; \]
\[ \text{MKAP} \cdots \text{MKAP}; \text{MKAP} \quad k \ell; \text{UPDATE} \quad (n + 1); \text{POP} \quad n; \]
\[ \text{UNWIND} \]
\[ C[(f_w e_1 \cdots e_m) \{ \text{-ROOTOPT } k \ell - \}] \quad r \; n = C[e_m] \quad r \quad n; \cdots; C[e_{k-l+1}] \quad r \quad n + m - (k - l - 1); \]
\[ \text{PUSH} \quad (n + m - (k - l) - r(\varsigma)); \text{MKAP} \cdots \text{MKAP}; \]
\[ \text{MKAP} \quad k \ell; \text{MKAP} \cdots \text{MKAP} \quad (m - k) \]

where \( \varsigma \) is a fresh variable and \( m \geq k \)

**Figure 11:** The compilation rules for worker/wrapper

\[ \langle o, \text{MKAP} \quad k \ell; c, n_1 : n_2 : s, G, E, D \rangle \Rightarrow \langle o, c, n'_1 : s, G[n'_1 = \text{AP}_{k, \ell} n_1 n_2], E, D \rangle \]

**Figure 12:** The transition rule for MKAP \( k \ell \)

### 3.3 A compilation example

By applying worker/wrapper transformation, \( \text{map} \) is transformed into the following two functions.

\[
\text{map} f \ell \quad = \quad (\text{map}_w f \ell) \{ \text{-BUILD 2 1-} \}
\]
\[
\text{map}_w f [] \quad = \quad []
\]
\[
\text{map}_w f (x : xs) \quad = \quad f x : (\text{map}_w f \; xs) \{ \text{-ROOTOPT 2 1-} \}
\]

The recursive instance \( \text{map}_w f \; xs \) in \( \text{worker}(\text{map}_w) \) and the body of \( \text{wrapper}(\text{map}) \) are compiled as following.

\[ C[\text{map}_w f \; xs \{ \text{-ROOTOPT 2 1-} \}] \quad r \; n = \quad \text{PUSH}(n - r(xs)); \text{PUSH}(n + 1 - r(\varsigma)); \text{MKAP} \quad 2 \; 1 \]

for some \( r, n \)

\[ \mathcal{R}[\text{map}_w f \; xs \{ \text{-BUILD 2 1-} \}] \quad r \; n = \quad \text{PUSH}(n - r(xs)); \text{PUSH}(n + 1 - r(f)); \]
\[ \text{PUSHGLOBAL} \; \text{map}_w; \text{MKAP}; \text{MKAP} \quad 2 \; 1; \]
\[ \text{UPDATE} \quad (n + 1); \text{POP} \; n; \text{UNWIND} \]

for some \( r, n \)

In the example, we can see that a partial application of the current redex is used by only \( \text{PUSH} \) instruction to construct a thunk over the recursive call instance.
4 Experiment

4.1 Implementation

We implement our idea on HBC, a Haskell compiler using G-machine. The overview of implementation is shown in Figure 13. G-code is the instruction of G-machine and M-code is the meta-level assembly language for convenience to generate machine codes. We modified the passes in double-lined boxes in HBC.

- An analysis finds a function to apply root optimization in syntactic concern, after lambda lifting. Then the function is transformed into worker and wrapper.

- Worker and wrapper are compiled by the new compilation rule shown in Figure 11.

- New runtime routines for new unwind instructions, i.e. \(\text{unwind}_{k,l}\), for \(ap_{k,l}\) are implemented.

To speed up unwind operation for each representation, it is useful to keep some specialized unwind routines for the most common arities[9]. Since the available tags should be known even before compile time in G-machine, we should determine which \((k,l)\) pairs for \(ap_{k,l}\) are used. After that, we should implement the specialized unwind routines, i.e., \(\text{unwind}_{k,l}\) for the pairs. We implement 28 pairs of \(\text{unwind}_{k,l}\) which are needed for all \textit{nofib} benchmark programs. The size of added runtime routines is 8,768 bytes. Although we prepare \(\text{unwind}_{k,l}\) for only finite number of pairs, we can avoid unsafety by not applying root optimization for the cases of unprepared pairs, if any. It is another issue which \((k,l)\) pairs may be used.

We turn on \texttt{-fno-zap-redux} not to overwrite the top application node during entering a function. The option makes somewhat faster, but sometimes leakier programs. Although our first implementation, used in our experiment, required not zapping a redex, we have revised the implementation to allow zapping a redex.

4.2 Results

We run the \textit{nofib} benchmark[16] compiled by the revised implementation of HBC on Sun Ultra Sparc I(Solaris v2.5) with 256Mb main memory. We analyze the number of applications, total heap consumption, and total execution time for the revised root optimization.

The statistics on the number of applications of root optimization are shown in Table 1. For comparison, the number of application of static argument transformation is also described.

![Figure 13: Revised implementation of root optimization on HBC](image-url)
<table>
<thead>
<tr>
<th>Program</th>
<th>Root optimization(a/b)</th>
<th>SAT(b)[17]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>exp3_8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>gen_regexps</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>paraffins</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>queens</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>HMMS</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>lift</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>pic</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>reptile</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(-) means that no data is available in [17]
SAT : static argument transformation
a : # of static args
b : # of args

Table 1: Applications of root opt. and static arg. transformation

in the same table. Note that G-machine requires applying lambda lifting to a given program, but STGM does not. By Table 1, we can see that root optimization is applied more frequently. It suggests that lambda lifting offers many chances to thunk sharing approach in G-machine. Sometimes, the translation of list comprehension offers more chances, incorporating lambda lifting.

Figure 14 shows a standard rule of list comprehension translation rules[12], which is used in HBC[2]. $\mathcal{TE}$ transforms an expression using list comprehension to a normal expression. A new recursive function $h$ is produced for every generator in list comprehension. Although $h$ has only one dynamic argument $xs$, it will have some static argument after lambda lifting, when it has some free variables. The functions for entries (10/11) and (12/13) of paraffins are due to above reason.

The statistics on the total heap consumption and the execution time without root optimization are shown in Table 2. We apply root optimization with the restriction on the number of dynamic arguments. For the benchmark programs shown in Figure 15, there is no

\[
\mathcal{TE}[E \mid P \leftarrow L, QS] \quad \rightarrow \quad [R] =
\]

\[
\text{let}
\]

\[
h \; xs = \text{case} \; xs \; \text{of}
\]

\[
[] \rightarrow \mathcal{R}
\]

\[
(x : xs') \rightarrow \mathcal{TE}[\text{case} \; x \; \text{of}
\]

\[
P \rightarrow [E \mid QS] + +h \; xs'
\]

\[
\rightarrow h \; xs'
\]

\[
in \quad h \; L
\]

Figure 14: A part of translation rule for list comprehension
function containing more than four dynamic arguments. Therefore, we apply our scheme with restriction, where the number of dynamic arguments are ≤1,≤2,≤3,and ≤4. The statistics for each restriction are depicted in Figure 15. Note that the case without root optimization uses the vap representation whenever possible instead of the ap representation. Most thunks tend to be constructed in the vap representation.

Large reduction in total heap allocation can be seen in exp3.8 and paraffins. It is due to the large number of recursive calls, e.g. 8,069,620 recursive calls for a function (1/2) in exp3.8. A large reduction of execution time happens also in paraffins. The paraffins shows that execution time may be reduced, if a large portion of an application is shared.

Unfortunately, most programs show almost no difference when root optimization is applied. Besides, in HMMS, total heap allocation is increased by 11.7% and execution-time is increased by 4.0%. In general, the overhead results from two reasons: the initial thunk creation overhead from wrapper and a chain of ap nodes over dynamic arguments of recursive call instances in worker. Gen_regexps suffers from the former because all the functions have only one dynamic argument. HMMS suffers from both reasons, but more from the second one. In Figure 15, the changes from ≤ 3 to ≤ 2 in HMMS can pinpoint the function (1/4). The function (1/4) is the main source of the overhead in HMMS. The ratio( # of outer calls to the function (1/4) ) is 1/18 and the ratio is sufficient to prove the second reason is severe by roughly calculating the number of entries of nodes.

- The initial overhead from wrapper is not avoidable in our implementation. A sufficient number of recursive calls should occur to compensate the initial overheads. Inlining wrapper to call sites may reduce the overhead.

- As the number of dynamic arguments is large, more ap nodes are allocated. Then both total heap allocation and execution time will be increased. If all the dynamic arguments were stored in a single contiguous block such as vap node, the overhead would be reduced. We have not implemented it yet, but we may enjoy the same effect when the number of dynamic arguments is only one because ap node is a single block containing all the dynamic arguments. Therefore, we may restrict applying root optimization to the function whose number of dynamic arguments is ≤ 1. Note that the result is acceptable under above restriction as observed in Figure 15.

The execution time of all benchmark programs is comparable in the original HBC judging by Figure 15. The revised root optimization does not always guarantee to run programs faster, but it may usually reduce the amount of heap usage with some acceptable execution

<table>
<thead>
<tr>
<th>program</th>
<th>HBC</th>
<th>program</th>
<th>HBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp3.8</td>
<td>129.2</td>
<td>gen_regexps</td>
<td>HMMS</td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td></td>
<td>74.6</td>
</tr>
<tr>
<td>paraffins</td>
<td>2.7</td>
<td></td>
<td>lift</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>queens</td>
<td>53.9</td>
<td></td>
<td>pic</td>
</tr>
<tr>
<td></td>
<td>5.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20.6</td>
<td></td>
<td>reptile</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

heap : total heap alloc.(Mbyte)
time : user+sys(sec.)

Table 2: The original performance
overhead. In our initial attempt to implement root optimization, we tried to experiment the original root optimization without any change on HBC and we got 8.2 sec. (+34.4%) as execution time of \textit{exp3.8}. This fact also supports the propriety of our implementation.

5 Related Works

Static argument transformation, which is a source-to-source transformation is implemented on GHC under STGM as an abstract machine [17]. It introduces \texttt{a let} which contains static arguments as nonlocal environment and makes a new internal function with the same behavior as the original, but different in accessing the static arguments. Here is an example of the transformation for \texttt{map}:

\begin{verbatim}
map f l = let map' [] = []
          map' (x:xs) = f x : map' xs
            in map' l
\end{verbatim}

A recursive call instance \texttt{map'} \texttt{xs} does not contain \texttt{f}, so that a thunk over the instance does not contain an entry for \texttt{f}. It is the key idea of the static argument transformation. In the
case with only one static argument, it cannot reduce heap usage. The closure for map is bound at compile-time since it contains no free variable, but the closure for map' is not since it contains a free variable f. Therefore, one entry for the address of the closure for map' must be exist in the thunk. Consequently, we can reduce heap usage only for the function with two or more static arguments[17].

Comparisons among static argument transformation, original root optimization, and the revised root optimization are shown in Table 3. The static argument transformation can applied together with other program transformations or tail call optimization. Also it can be easily adapted to mutual recursive functions, too. As our revised root optimization, it needs a sufficient number of recursive calls and it does not depend on the position of static arguments. The original root optimization can be applied differently from each recursive call instance which may have the different number of static arguments.

For completeness, we summarize other related works. A.W.Appel suggests similar one to static argument transformation for SML-NJ compiler to expose inlining recursive functions[1]. O.Danvy suggests lambda dropping to restore the original block structure of lambda-lifted program[4]. Static argument transformation is used to preserve laziness in deforestation[15] and to make the application of strictness analysis more frequently[18].

### 6 Conclusion and Further Work

We suggest a practical implementation of root optimization using new tags and specialized runtime routines to overlap the unwind operation and root operation. We apply the worker and wrapper transformation to allow to use freely both the vap representation and tail call optimization. Because the vap representation is more efficient thunk representation and tail call optimization is an important method for call-intensive language like functional language, they should not be abandoned.

We implement our idea on HBC and we confirm our claim that our implementation can reduce heap usages with the acceptable execution-time. We also suggest heuristic that root optimization is applied only when the number of dynamic arguments is one. Under this strategy, we avoid overhead with a chain of dynamic arguments and we can apply root optimization fully to the function generated during the translation of list comprehension. We
believe our implementation is worthwhile as an optimizing pass.
Our idea can be improved in some respects:

- The overhead in wrapper can be reduced by inlining the wrapper to an outer call, so that a thunk construction over outer call, if any, should not be useless immediately.

- The overhead in worker can be reduced by representing a thunk with two blocks: one for all the dynamic arguments and the other for all the static arguments.

- By the conversion of mutual recursive function into direct one[14], root optimization can be applied easily to mutual recursive functions.

References


