Vol. 47 No. SIG 6(PRO 29) IPSJ Transactions on Programming May 2006

Regular Paper

A Transformation-Based Implementation of Lightweight Nested Functions

Tasuku Hiraishi,[†] Masahiro Yasugi[†] and Taiichi Yuasa[†]

The SC language system was developed to provide a transformation-based language extension scheme for SC languages (extended/plain C languages with an S-expression-based syntax). Using this system, many flexible extensions to the C language can be implemented by means of transformation rules over S-expressions at low cost, mainly because of the preexisting Common Lisp capabilities for manipulating S-expressions. This paper presents the LW-SC (Lightweight-SC) language as an important application of this system, featuring nested functions (i.e., functions defined inside other functions). A function can manipulate its caller's local variables (or local variables of its indirect callers) by indirectly calling a nested function of its callers. Thus, many high-level services with "stack walk" can be easily and elegantly implemented by using LW-SC as an intermediate language. Moreover, such services can be implemented efficiently because we designed and implemented LW-SC to provide "lightweight" nested functions by aggressively reducing the costs of creating and maintaining nested functions. The GNU C compiler also provides nested functions as an extension to C, but our sophisticated translator to standard C is more portable and efficient for occasional "stack walk."

1. Introduction

The C language is often indispensable for developing practical systems. Furthermore, extended C languages are sometimes suitable for elegant and efficient development. A language extension can be implemented by modifying a C compiler, but in some cases it can also be done by translating an extended C program into C. We developed the SC language system[8),10)] to facilitate such transformation-based language extensions. SC languages are extended/plain C languages with an S-expression-based syntax, whose extensions are implemented by transformation rules over S-expressions. Thus it is possible to reduce implementation costs mainly because of the ease with which S-expressions can be manipulated by using Lisp.

The fact that C has low-level operations (e.g., pointer operations) enables many flexible extensions to be implemented by using the SC language system. But without taking "memory" addresses, C lacks an ability to access variables sleeping in the execution stack, which is required for implementation of high-level services with "stack walk" such as capturing a stack state for check-pointing and scanning roots for copying GC (garbage collection).

A possible solution to this problem is to support *nested functions*. A nested function is a function defined inside another function. A function can manipulate its caller's local variables (or local variables of its indirect callers) sleeping in the execution stack by indirectly calling a nested function of its callers.

This paper presents an implementation of an extended SC language, named LW-SC (Lightweight SC), which features nested functions[☆]. Many of the above-mentioned high-level services with "stack walk" can be easily and elegantly implemented by using LW-SC as an intermediate language. Moreover, such services can be implemented efficiently because we designed and implemented LW-SC to provide "lightweight" nested functions by aggressively reducing the costs of creating and maintaining nested functions. Though the GNU C compiler[15)] (GCC) also provides nested functions as an extension to C, our sophisticated translator to standard C is more portable and efficient for occasional "stack walk."

Note that, though this paper presents an implementation using the SC language system, our technique is not limited to it.

† Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

☆ We have previously reported the implementation of LW-SC as an example of a language extension using the SC language system[10)]. This paper discusses further details of LW-SC itself.

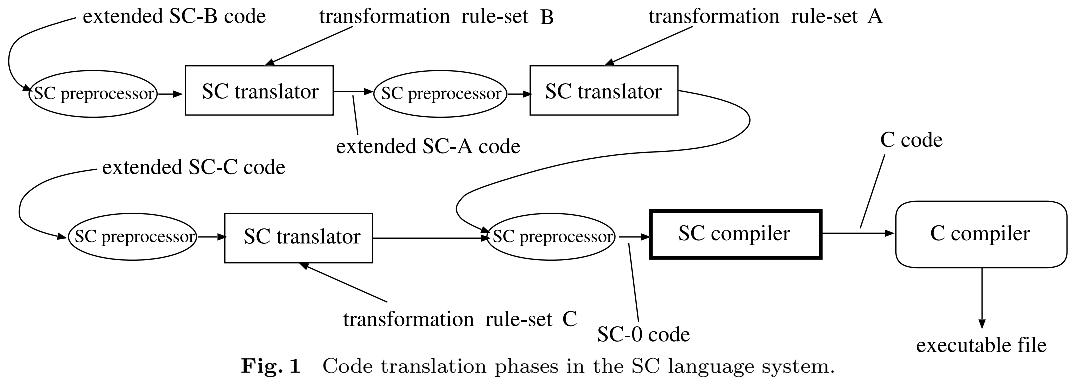


Fig. 1 Code translation phases in the SC language system.

2. The SC Language System

This section explains those aspects of the SC language system necessary for the specification and implementation of LW-SC. More details are available in our past papers[8),10)].

2.1 Overview

The SC language system, implemented in Common Lisp, deals with the following S-expression-based languages:

- SC-0, the base SC language, and
- extended SC languages,

and consists of the following three modules:

- the SC preprocessor,which includes SC files and handles macro definitions and expansions,
- the SC translator,which interprets transformation rules for translating SC code into another SC, and
- the SC compiler,which compiles SC-0 code into C.

Figure 1 shows code translation phases in the SC language system. Extended SC code is translated into SC-0 by the SC translators, and then translated into C by the SC compiler. Before each translation phase with a transformation rule-set is applied, preprocessing by the SC preprocessor is performed. Extension implementers can develop a new translation phase simply by writing new transformation rules.

2.2 The SC Preprocessor

The SC preprocessor handles the following SC preprocessing directives to transform SC programs:

- `(%include` *file-name*`)`
  corresponds to an `#include` directive in C. The file *file-name* is included.
- `(%defmacro` *macro-name lambda-list*
  S-expression$_1 \cdots$ *S-expression*$_n$`)`
  evaluated as a `defmacro` form of Common Lisp to define an SC macro. After the definition, every list in the form of `(`*macro-name* $\cdots$`)` is replaced with the result of the application of Common Lisp's `macroexpand-1` function to the list. The algorithm to expand nested macro applications complies with the standard C specification.
- `(%defconstant` *macro-name S-expression*`)`
  defines an SC macro in the same way as a `%defmacro` directive, except that every symbol which is `eq` to *macro-name* is replaced with *S-expression* after the definition.
- `(%undef` *macro-name*`)`
  undefines the specified macro defined by `%defmacro`s or `%defconstant`s.
- `(%ifdef` *symbol list*$_1$ *list*$_2$`)`
  `(%ifndef` *symbol list*$_1$ *list*$_2$`)`
  If the macro specified by *symbol* is defined, *list*$_1$ is spliced there. Otherwise, *list*$_2$ is spliced.
- `(%if` *S-expression list*$_1$ *list*$_2$`)`
  S-expression is macro-expanded, and then the result is evaluated by Common Lisp. If the return value is `eql` to `nil` or `0`, *list*$_2$ is spliced there. Otherwise, *list*$_1$ is spliced.
- `(%error` *string*`)`
  interrupts the compilation with an error message *string*.
- `(%cinclude` *file-name*`)`
  file-name specifies a C header file. The C code is compiled into SC-0 and the result is spliced there. SC programmers can use library functions and most of the macros, such as `printf` and `NULL`, declared/`#define`d in C header files[☆].

☆ In some cases such a translation is not obvious. In particular, it is sometimes impossible to translate `#define` macro definitions into `%defmacro` or `%defconstant`. We have discussed this problem elsewhere[9)].

2.3 The SC Translator and Transformation Rules

A transformation rule for the SC translator is given by the syntax:

(*function-name pattern parm*$_2 \cdots$ *parm*$_n$)
-> *expression*,

where a function *function-name* is defined as an ordinary Lisp function. When the function is called, the first argument is tested to determine whether it matches *pattern*. If it matches, *expression* is evaluated by the Common Lisp system, and its value is then returned as the result of the function call. The parameters *parm*$_2 \cdots$ *parm*$_n$, if any, are treated as ordinary arguments.

A list of transformation rules may include two or more rules with the same function name. In these cases, the first argument is tested to determine whether it matches each *pattern*, in written order, and the result of the function call is the value of *expression* if there is a match.

It is permissible to abbreviate

(*function-name pattern*$_1$ *parm*$_2 \cdots$ *parm*$_n$)
-> *expression*
...
(*function-name pattern*$_m$ *parm*$_2 \cdots$ *parm*$_n$)
-> *expression*

(all the *expressions* are identical and only *patterns* are different from each other) to

(*function-name pattern*$_1$ *parm*$_2 \cdots$ *parm*$_n$)
...
(*function-name pattern*$_m$ *parm*$_2 \cdots$ *parm*$_n$)
-> *expression*.

The *pattern* is an S-expression consisting of one of the following elements:

(1) *symbol*
matches a symbol that is `eq` to *symbol*.

(2) `,`*symbol*
matches any S-expression.

(3) `,@`*symbol*
matches any list of elements longer than 0.

(4) `,`*symbol*`[`*function-name*`]`
matches an *element* if the evaluation result of `(funcall #'`*function-name element*`)` is non-`nil`.

(5) `,@`*symbol*`[`*function-name*`]`
matches *list* (longer than 0) if the evaluation result of `(every #'`*function-name list*`)` is non-`nil`.

The function *function-name* can be what is defined as a list of transformation rules or an ordinary Common Lisp function (a built-in function or a function defined separately from transformation rules).

In evaluating *expression*, the special variable `x` is bound to the whole matched S-expression and, in all the cases except (1), *symbol* is bound to the matched part in the S-expression.

An example of such a function definition can be given as follows[☆]:

```
(EX (,a[numberp] ,b[numberp]))
  -> `(,a ,b ,(+ a b))
(EX (,a ,b))
  -> `(,a ,b ,a ,b)
(EX (,a ,b ,@rem))
  -> rem
(EX ,otherwise)
  -> '(error)
```

The application of the function `EX` can be exemplified as follows:

```
(EX '(3 8))     → (3 8 11)
(EX '(x 8))     → (x 8 x 8)
(EX 8)          → (error)
(EX '(3))       → (error)
(EX '(x y z))   → (z)
```

Each set of transformation rules defines one or more (in most cases) function(s). A piece of extended SC code is passed to one of the functions, which generates transformed code as the result.

Internally, transformation rules for a function are compiled into an ordinary Common Lisp function definition (`defun`). The output can be loaded by the `load` function, which makes it easy for programmers to test some parts of transformation rule-sets in an interactive environment.

2.4 The SC Compiler and the SC-0 Language

We designed the SC-0 language as the final

[☆] In consideration of symmetry between expressions and patterns, it is more pertinent to describe `` `(,a[numberp] ,b[numberp]) `` with a backquote. However, this notation rule leads to the inconvenience that programmers have to put backquotes before most patterns. We preferred shorter descriptions, and have therefore adopted the notation without backquotes.

```
(def (sum a n) (fn int (ptr int) int)
  (def s int 0)
  (def i int 0)
  (do-while 1
    (if (>= i n) (break))
    (+= s (aref a (inc i))))
  (return s))
```

Fig. 2 An SC-0 program.

```
int sum (int* a, int n)  {
  int s=0;
  int i=0;
  do{
   if ( i >= n ) break;
   s += a[i++];
  } while(1);
  return s;
}
```

Fig. 3 C program equivalent to Fig. 2.

target language for translation by transformation rules. It has the following features:

- an S-expression-based, Lisp-like syntax,
- the C semantics; actually most C code can be represented in SC-0 in a straightforward manner☆, and
- practicality for programming.

Figure 2 shows an example of such an SC-0 program, which is equivalent to the program in **Fig. 3**.

In practice, the SC compiler is implemented as a transformation rule-set described above, which specifies transformation from S-expressions into strings (instead of S-expressions).

3. Language Specification of LW-SC

LW-SC has the following features as extensions to SC-0:

- **Nested function types:**
  `(lightweight` *type-expression-list*`)`
  is added to the syntax for *type-expression.*
- **Calling nested functions:** In function-call expressions (`(`*expression-list*`)`), the type of the first expression is permitted to be any nested function pointer type other than the ordinary function pointer type.
- **Defining nested functions:** In places where variable definitions are allowed, except at the top level, definitions of nested functions are permitted in the following form:
  `(def (`*identifier-list*`)`
  `(lightweight` *type-expression-list*`)`
  block-item-list`)`
  (the syntax is almost the same as that of ordinary function definitions, except for the difference between the keywords `fn` and `lightweight`.)

☆ Except for some features such as `->` operators, `for` constructs, and `while` constructs. These are implemented as language extensions to SC-0 using the SC language system itself.

```
(def (h i g) (fn int int (ptr (lightweight int int)))
  (return (g (g i))))

(def (foo a) (fn int int)
  (def x int 0)
  (def y int 0)
  (def (g1 b) (lightweight int int)
    (inc x)
    (return (+ a b)))
  (= y (h 10 g1))
  (return (+ x y)))

(def (main) (fn int)
  (return (foo 1))
```

Fig. 4 An LW-SC program.

A nested function can access the lexically scoped variables in the creation-time environment and a pointer to it can be used as a function pointer to indirectly call the closure. For example, **Fig. 4** shows an LW-SC program. When `h` indirectly calls the nested function `g1`, it can access a parameter `a` and local variables `x`, `y` sleeping in `foo`'s frame.

4. GCC's Implementation of Nested Functions

GCC also features nested functions, and the specification of nested functions in LW-SC is almost the same as that in GCC. As in GCC (but unlike closure objects in modern languages such as Lisp and ML), nested functions of LW-SC are valid only when the owner blocks are alive. But unlike GCC, pointers to nested functions are not compatible with those to top-level functions. However, such limitations are insignificant for the purpose of implementing most high-level services with the "stack walk" mentioned in Section 1.

GCC's implementation of nested functions causes high maintenance/creation costs, for the following reasons:

- In creating nested functions, there is a cost for initializing. To initialize an address-taken nested function, GCC uses a technique called *trampolines*[2]. A trampoline is a code fragment generated on the stack at runtime to indirectly enter the nested function with a necessary environment. The cost of runtime code generation is high, and for some processors such as SPARC, it is necessary to flush some instruction caches for the runtime-generated trampoline code.

- Local variables and parameters of a function generally may be assigned to registers if the function contains no nested function. But an owner function of GCC's nested functions must keep the values of these variables in the stack so that the nested functions can access them, usually via a static chain. Thus, the owner function must perform memory operations to access these variables, which means that the cost of maintaining nested functions is high.

LW-SC overcomes the former problem by translating the nested function into a lazily initialized pair (on the explicit stack) of the ordinary function pointer and the frame pointer, and the latter by saving the local variables to the "explicit stack" lazily (only on calls to nested functions), as is shown in the following section.

5. Implementation of LW-SC

We implemented LW-SC described above by using the SC language system, that is, by writing transformation rules for translation into SC-0, which is finally translated into C.

5.1 Basic Ideas

The basic ideas for implementing nested functions by translation are summarized as follows:

- After transformation, all definitions of nested functions can be moved to be top-level definitions.
- To enable the nested functions to access local variables of their owner functions, an explicit stack is employed in C other than the (implicit) execution stack for C. The explicit stack mirrors values of local variables in the execution stack, and is referred to when local variables of the owner functions are accessed.
- To reduce the costs of creating and maintaining nested functions, operations to fix inconsistency between two stacks are delayed until nested functions are actually invoked.

Function calls/returns and function definitions in LW-SC should be appropriately transformed in accordance with these ideas.

5.2 Transformation

LW-SC programs are translated in the following way to realize the ideas described in Section 5.1.

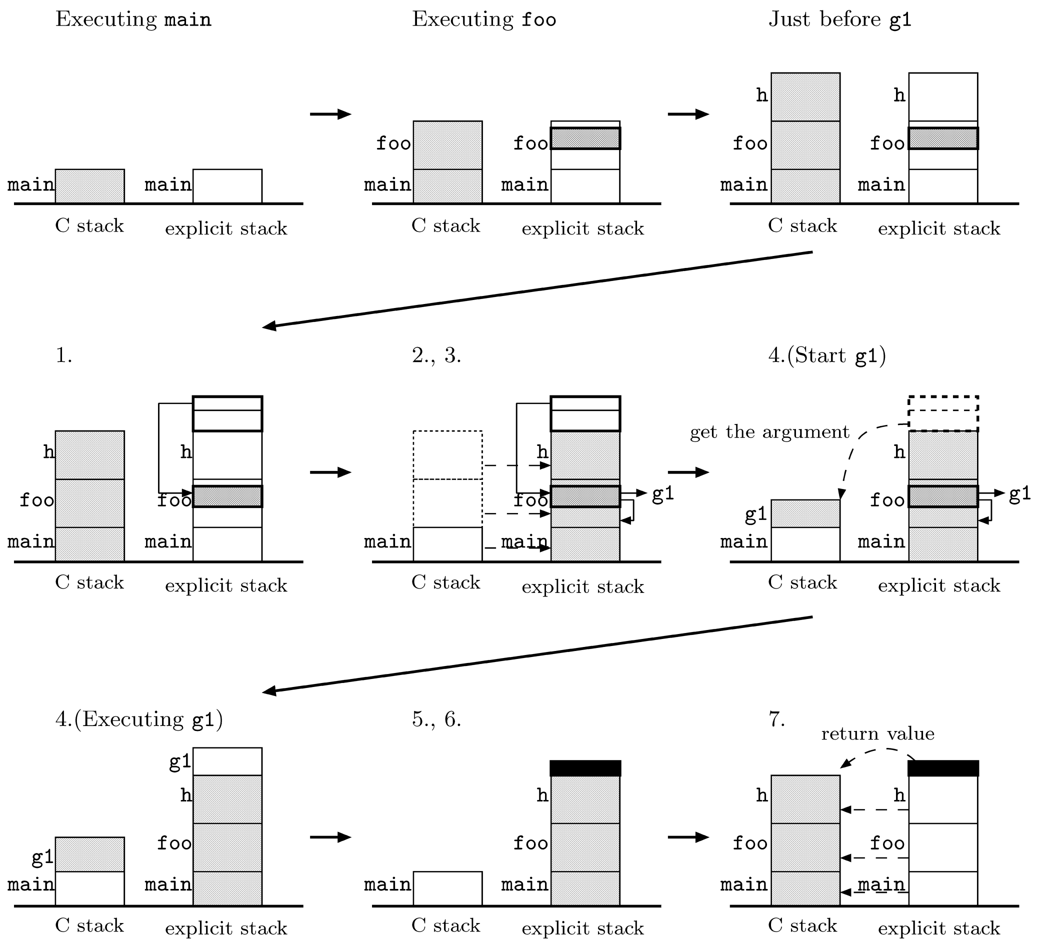
(a) Each generated C program employs an explicit stack of the type mentioned above in memory. This shows a logical execution stack, which manages local variables, callee frame pointers, arguments, return values of nested functions (of LW-SC), and return addresses.

(b) Each function call to an ordinary top-level function in LW-SC is transformed into the same function call in C, except that a special argument is added which saves the stack pointer to the explicit stack. The callee first initializes its frame pointer with the stack pointer, next shifts the stack pointer by its frame size, and then executes its body.

(c) Each nested function definition in LW-SC is moved to the top level in C. Instead, a structure object, which contains the pointer to the nested function that was moved and the frame pointer of the owner function, is stored on the explicit stack. Note that initialization of the structure is delayed until nested functions are invoked to reduce the costs of creating nested functions.

(d) Each function call to a nested function in LW-SC is translated into the following steps:

1. Push arguments passed to the nested function and the pointer to the structure mentioned above in (c) to the explicit stack.
2. Save the values of the all local variables and parameters, and an integer corresponding to the current execution point (return address), into the explicit stack, then return from the function.
3. Iterate Step 2 until control is returned to `main`. The values of local variables and parameters of `main` are also stored on the explicit stack.
4. Referring to the structure which is pointed to by the pointer pushed at Step 1 (the one in (c)), call the nested function whose definition has been moved to the top level in C. The callee first obtains its arguments by popping the values pushed at Step 1, then executes its body.
5. Before returning from the nested function, push the return value to the explicit stack.
6. Reconstruct the execution stack by restoring the local variables, the parameters, and the execution points, with the values saved in the explicit stack



▨ : Stack memory that contains the correct values of local variables.
▨ : Structure that contains a pointer to function `g1` (moved to the top level) and a pointer to `foo`'s frame.
□ : Arguments to `g1` and the pointer to the above structure.
■ : Return value of `g1`.

Fig. 5 Details of an indirect call to a nested function `g1` in Fig. 4.

at Step 3 (the values may be changed during the call to the nested function), to return to (resume) the caller of the nested function.

7. If necessary, get the return value of the nested function pushed at Step 5 by popping the explicit stack.

Note that a callee (a nested function) can access the local variables of its owner functions through the frame pointers contained in the structure that have been saved at Step 1.

For example, **Fig. 5** shows the state transition of the two stacks☆, in the case of Fig. 4, from the beginning of the execution until the end of the first indirect call to a nested function `g1`. (Each number in the figure corresponds to the step of the nested function call described in (d).) Notice that the correct values of the local variables are saved in the explicit stack during the execution of the nested function and in the C stack at other times.

☆ "The C stack" here simply contains the set of local variables and parameters, whose values are stored not only in the stack memory but also in registers.

5.3 Transformation Rules

To implement the transformation described above, we wrote transformation rules. The entire transformation is divided into the following four phases (rule-sets) for simplicity and reusability of each phase:

(1) **The `type` rule-set:** adds type information to all the *expression*s of an input program.
(2) **The `temp` rule-set:** transforms an input program in such a way that no function call appears as a subexpression (except as a right-hand side of an assignment).
(3) **The `lightweight` rule-set:** performs the transformation described in Section 5.2.
(4) **The `untype` rule-set:** removes the type information added by the `type` rule-set from *expression*s to generate correct SC-0 code.

The following subsections present the details of these transformation rule-sets.

5.3.1 The `type` rule-set

Transformation by the `temp` rule-set and the `lightweight` rule-set requires type information on all expressions. The `type` rule-set adds such information. More specifically, it transforms each *expression* into `(the` *type-expression expression*`)`.

Figure 6 shows the (abbreviated) transformation rule-set. `Tp0` is applied to an input program (e.g., that in **Fig. 7**) to obtain a transformed program (e.g., that in **Fig. 8**). `Tp1` receives declarations and renews the dynamic variables which save the information about defined variables, structures, etc. `Tpe` actually transforms expressions referring to the dynamic variables.

5.3.2 The `temp` rule-set

A function call appearing as a subexpression such as `(g x)` in `(f (g x))` makes it difficult to add some operations just before/after the function call. The `temp` rule-set prevents such function calls from appearing.

Because some temporary variables are needed for the transformation, their definitions are inserted at the head of the function body. For example, the program in **Fig. 9** is transformed into the program in **Fig. 10** by means of this rule-set.

Figure 11 shows the (abbreviated) `temp` rule-set. The actual transformation is performed by `Tmpe`, which returns a 3-tuple of

- a list of the variable definitions to be inserted at the head of the current function,
- a list of the assignments to be inserted just before the expression, and
- an expression with which the current expression should be replaced.

`Tmp` and `Tmp2` combine the tuples appropriately and finally `Tmp0` returns the transformed code.

```
(Tp0 (,@declaration-list) )
-> (progn
    ...
    (let (*str-alist* *v-alist* *lastv-alist*)
     (mapcar #'Tp1 declaration-list)))
;;;;; declaration ;;;;;
(Tp1 (,scs[SC-SPEC] ,id[ID] ,texp ,@init))
-> (progn
    (push (cons id (remove-type-qualifier texp))
          *v-alist*)
     `(,scs ,id ,texp ,@(mapcar #'Tpi init)))
(Tp1 (,scs[SC-SPEC] (,@id-list[ID])
      (fn ,@texp-list) ,@body))
-> (let* ((texp-list2
           (mapcar #'rmv-tqualifier texp-list))
          (*v-alist* (cons (cons (first id-list)
                          `(ptr (fn ,@texp-list2)))
                               *v-alist*))
          (new-body nil))
    (let ((b-list
           (cmpd-list (cdr id-list)
                      (cdr texp-list2))))
     (setq new-body
      (let((*v-alist* (append b-list *v-alist*))
           (*str-alist* *str-alist*))
        (mapcar #'Tpb body))))
    `(,scs (,@id-list)
      (fn ,@texp-list),@new-body))
...
(Tp1 ,otherwise)
-> (error "syntax error")
;;;;;;; body ;;;;;;
(Tpb (do-while ,exp ,@body))
-> (switch ,(Tpe exp)
     ,@(let ((*v-alist* *v-alist*)
             (*str-alist* *str-alist*))
         (mapcar #'Tpb body)))
...
(Tpb ,otherwise)
-> (let ((expression-stat (Tpe otherwise)))
     (if (eq '$not-expression expression-stat)
         (Tp1 otherwise)
       expression-stat))
;;;;; expression ;;;;;
(Tpe ,id[ID])
-> `(the ,(assoc-vartype id) ,id)
...
(Tpe (ptr ,exp))
-> (let ((exp-with-type (Tpe exp)))
      `(the (ptr ,(cadr exp-with-type))
        (ptr ,exp-with-type)))
(Tpe (mref ,exp))
-> (let* ((exp-with-type (Tpe exp))
          (exp-type (cadr exp-with-type)))
     `(the ,(deref-type exp-type)
       (mref ,exp-with-type)))
(Tpe (,fexp[EXPRESSION] ,@arg-list))
-> (let* ((fexp-with-type (Tpe fexp))
          (fexp-type (cadr fexp-with-type))
          (type-fn (cadr fexp-type)))
     `(the ,(cadr type-fn)
       (call (the ,type-fn
               ,(caddr fexp-with-type))
        ,@(mapcar #'Tpe arg-list))))
(Tpe ,otherwise)
-> '$not-expression
```

Fig. 6 The `type` rule-set (abbreviated).

```
(def (g x) (fn int int)
  (return (* x x)))

(def (f x) (fn double double)
  (return (+ x x)))

(def (h x) (fn char double)
  (return (f (g x))))
```

Fig. 7 Example of a program to which the `type` rule-set is applied (before transformation).

```
(def (g x) (fn int int)
  (return (the int
             (* (the int x) (the int x)))))

(def (f x) (fn double double)
  (return (the double
             (+ (the double x) (the double x)))))

(def (h x) (fn char double)
  (return (the double
             (call (the (fn double double) f)
                   (the int (call (the (fn int int) g)
                                  (the double x)))))))
```

Fig. 8 Example of a program to which the `type` rule-set is applied (after transformation).

```
(def (g x) (fn int int)
  (return
     (the int
          (+ (the int
                  (= (the int x) (the int 3)))
                  (the int
                       (call (the (fn int int) g)
                             (the int x)))))))
```

Fig. 9 Example of a program to which the `temp` rule-set is applied (before transformation).

```
(def (g x) (fn int int)
  (def tmp1 int)
  (def tmp2 int)
  (the int
       (= (the int tmp1)
       (the int
            (= (the int x) (the int 3)))))
  (the int
       (= (the int tmp2)
          (the int
               (call (the (fn int int) g)
               (the int x)))))
  (return
    (the int
         (+ (the int tmp1) (the int tmp2)))))
```

Fig. 10 Example of a program to which the `temp` rule-set is applied (after transformation).

```
(Tmp0 (,@decl-list))
->(progn
    ...
    (let ((*used-id* (get-all-id x))
          (*prev-continue* nil))
      (mapcar #'Tmp1 x)))
;;;; declaration ;;;;;
(Tmp1 (,scs[SC-SPEC]
       (,@id-list[ID]) (fn ,@texp-list) ,@body))
-> (let* ((tmpbody (Tmp2 body))
          (newdecl (first tmpbody))
          (newbody (second tmpbody)))
    '(,scs (,@id-list)
      (,fntag ,@texp-list) ,@newdecl ,@newbody))
...
;;;;; body ;;;;;
(Tmp2 (,@item-list))
-> (let* ((tmpitemlist (mapcar #'Tmp item-list))
          (decl-list (apply #'append
                 (mapcar #'first tmpitemlist)))
          (prev-stat (apply #'append (mapcar
       #'(lambda (x) '(,@(second x) ,(third x)))
                                  tmpitemlist))))
     (list decl-list prev-stat))
(Tmp (do-while ,exp ,@body))
->(let* ((tmpexp (Tmpe exp))
         (*prev-continue* (second tmpexp))
         (tmpbody (Tmp2 body)))
   (list (append (first tmpexp) (first tmpbody))
         nil
    (do-while ,(third tmpexp)
         ,@(second tmpbody) ,@*prev-continue*)))
(Tmp (return ,@exp))
-> (if (null exp)
       '(nil nil (return))
       (let ((tmpexp (Tmpe (car exp))))
         '(,(first tmpexp) ,(second tmpexp)
           (return ,(third tmpexp)))))
...
(Tmp ,otherwise)
->(let ((tmpe-exp (Tmpe otherwise)))
    (if (eq '$not-expression tmpe-exp)
        (list (list (Tmp1 otherwise)) nil)
        tmpe-exp))
;;;;; expression ;;;;;
(Tmpe (the ,texp (call ,fexp ,@arg-list)))
-> (case texp
     ((void)
      ...  )
 (otherwise
 (let*
 ((tmpexps (comb-list (mapcar #'Tmpe arg-list)))
  (tempid (generate-id "tmp"))
  (tmp-decl1 '(def ,tempid ,texp))
  (tmp-decl
   (append (first tmpexps) '(,tmp-decl1)))
  (tmp-set1 '(the ,texp (= (the ,texp ,tempid)
   (the ,texp (call ,fexp ,@(third tmpexps))))))
  (tmp-set
   (append (second tmpexps) '(,tmp-set1))))
 (list tmp-decl tmp-set '(the ,texp ,tempid)))))
(Tmpe (the ,texp (+ ,exp1 ,exp2)))
-> (let ((op (caaddr x))
    (t-exp1 (Tmpe exp1)) (t-exp2 (Tmpe exp2)))
 (list '(,@(first t-exp1) ,@(first t-exp2))
       '(,@(second t-exp1) ,@(second t-exp2))
       '(the ,texp (,op ,(third t-exp1)
                   ,(third t-exp2)))))
...
```

Fig. 11 The `temp` rule-set (abbreviated).

5.3.3 The `lightweight` rule-set

Now the transformation described in Section 5.2 is realized by the `lightweight` rule-set. **Figure 12** shows the (abbreviated) `lightweight` rule-set which is related to the transformation of "ordinary function" calls and "nested function" calls. In the code, `esp` is a special parameter added to each function, which keeps the stack top of the explicit stack. `Efp` is a special local variable added to each function, which acts as the (explicit) frame pointer of the function. `Lwe-xfp` transforms accesses to local variables into accesses to the

```
;;;; Due to the temp rule-set, a function call expression must appear in either of the following forms
;;;; as a statement expression:
;;;;  * (= variable function-call-expression)
;;;;  * (= function-call-expression).

;;; "Ordinary function" call
(Lwe (the ,texp0 (= (the ,texp1 ,id) (the ,texp (call (the (fn ,@texp-list) ,exp-f) ,@exp-list)))))
(Lwe                                 (the ,texp (call (the (fn ,@texp-list) ,exp-f) ,@exp-list)))
-> (let (...)
     ...
     (let* (...)
       (list nil decl-list
             (cons '(= new-esp esp) prev-list)
             '(while '(and (== (= ,(Lwe-xfp '(the ,texp1 ,id))
                                  (call ,fexp new-esp ,@(cdr tmpid-list)))
                              (special ,texp0))
                           (!= (= (fref efp -> tmp-esp) (mref-t (ptr char) esp))))
               ;; Save the values of local variables to the frame.
               ,@(make-frame-save *current-func*)
               ...
               ;; Save the current execution point.
               (= (fref efp -> call-id)
                ,(length (finfo-label-list *current-func*)))
               ;; Return from the current function
               ;; (In main, call the nested function here instead of taking the following steps).
               ,(make-suspend-return *current-func*)
               ;; Continue the execution from here when reconstructing the execution stack.
               (label ,(caar (push (cons (generate-id "L_call" *used-id-list*) nil)
                                   (finfo-label-list *current-func*)))
                nil)
               ;; Restore local variables from the explicit stack.
               ,@(make-frame-resume *current-func*)
               ...
               (= new-esp (+ esp 1))))))

;;; "Nested function" call
(Lwe (the ,texp0 (= (the ,texp1 ,id) (the ,texp (call (the (lightweight ,@texp-list) ,exp-f) ,@exp-list)))))
(Lwe                                 (the ,texp (call (the (lightweight ,@texp-list) ,exp-f) ,@exp-list)))
-> (let (...)
     ...
     (list '() fp-decl '()
           '(begin
             ...
             (= argp (aligned-add esp (sizeof (ptr char))))
             ;; Push the arguments passed to the nested function
             ,@(mapcar (compose #'(lambda (x) '(push-arg ,(second x) ,(third x) argp))
                                 #'Lwe-xfp)
                       (reverse exp-list))
             ;; Push the structure object that corresponds to the frame of the nested function to the explicit stack.
             (= (mref-t (ptr closure-t) argp) ,xfp-exp-f)
             ...
             ;; Save the values of local variables to the frame.
             ,@(make-frame-save *current-func*)
             (= (fref efp -> argp) argp)
             (= (fref efp -> tmp-esp) argp)
             ;; Save the current execution point.
             (= (fref efp -> call-id)
                ,(length (finfo-label-list *current-func*)))
             ;; Return from the current function (In main, call the nested function here instead of the following steps).
             ,(make-suspend-return *current-func*)
             ;; Continue the execution from here after the function call finishes.
             (label ,(caar (push (cons (generate-id "L_call" *used-id-list*) nil)
                                 (finfo-label-list *current-func*)))
                    nil)
             ;; Restore local variables from the explicit stack.
             ,@(make-frame-resume *current-func*)
             ;; Get the return value (if necessary).
             ,@(when assign-p
                 '( (= ,(Lwe-xfp '(the ,texp1 ,id))
                       (mref-t ,texp1 (fref efp -> argp))) )) )))
```

Fig. 12 The `lightweight` rule-set (abbreviated).

explicit stack.

"Ordinary function" calls and "nested function" calls can be statically distinguished by the functions' types, because ordinary function types are incompatible with lightweight nested function types.

The transformation of each operation is performed as follows (the rules unrelated to function calls are omitted in the figure):

Calling ordinary functions: The function call is performed as a part of the conditional expression of the `while` statement, where the stack pointer is passed to the callee as an additional first argument. If the callee procedure finished normally, the condition becomes false and the body of the `while` loop is not executed. Otherwise, if the callee returned for a "nested function" call, the condition becomes true. In the body of the `while` loop, the values of local variables are saved to the explicit stack, an integer that corresponds to the current execution point is also saved to the explicit stack ((fref efp -> call-id)), and then the current function temporarily exits. This function is re-called in order to reconstruct the execution stack after the execution of the nested function. The control is then transferred to the label placed next to the return by a goto statement, which is added in the head of the function. The values of local variables are then restored from the explicit stack and the function call in the conditional expression of the while statement is restarted. The assignment (= new-esp (+ esp 1)) at the end of the while block sets a flag at the LSB of the explicit stack pointer that indicates reconstruction of the execution stack.

Calling nested functions: The arguments passed to the nested function and the closure structure (which contains the nested function pointer and the frame pointer of its owner function) are pushed to the explicit stack. Then, as in an "ordinary function" call, the values of local variables and the executing point are saved, the current function exits, and the execution point is restored by goto after the steps for calling the nested function. The values of local variables are then restored and the return value of the nested function, if one exists, is taken from the top of the explicit stack.

Returning from functions: Returns from ordinary function need no transformation. On the other hand, returns from nested functions must be transformed in order to push the return value to the explicit stack, and simply to return 0 so as to indicate that the execution of the function finished normally.

Function definitions: The following steps are added before the functions' body:

- initialization of the frame pointer of the explicit stack (efp) and the stack pointer (esp),
- judgment as to whether reconstruction of the execution stack is required or not and, if it is required, execution of goto to the label corresponding to (fref efp -> call-id), and
- popping of parameters from the explicit stack, in the case of nested functions.

The transformation also involves adding the parameter esp that receives the explicit stack pointer, adding some local variable definitions, and adding a structure definition that represents the function's frame in the explicit stack and is referred to by efp.

```
(UTp0 ,decl-list)
-> (UTp decl-list)
(UTp (the ,texp ,exp))
-> (Utp exp)
(UTp (call ,@exp-list))
-> (mapcar #'Utp exp-list)
(UTp (,@lst))
-> (mapcar #'UTp lst)
(UTp ,otherwise)
-> otherwise
```

Fig. 13 The untype rule-set.

5.3.4 The untype rule-set

The output code transformed by the lightweight rule-set is not valid SC-0 code because it contains type information. The untype rule-set removes such information and generates valid SC-0 code. The rule-set is very simple; it only needs to search (the ...) forms recursively and to remove type information. **Figure 13** shows the entire untype rule-set.

As an example of a total translation, Appendix shows the entire SC-0 code generated from the LW-SC program in Fig. 4.

6. Evaluation

6.1 Creation and Maintenance Cost

To measure the costs of creating and maintaining nested functions, we employed the following programs with nested functions for several high-level services and compared them with the corresponding plain C programs:

BinTree (copying GC) creates a binary search tree with 200,000 nodes, with a copying-collected heap (**Fig. 14**).

Bin2List (copying GC) converts a binary tree with 500,000 nodes into a linear list, with a copying-collected heap (**Fig. 15**).

fib(34) (check-pointing) calculates the 34th Fibonacci number recursively, with a capability for capturing a stack state for check-pointing (**Fig. 16**).

nqueens(13) (load balancing) solves the N-queens problem (N=13) on a load-balancing framework based on lazy partitioning of sequential programs 21),22).

Note that nested functions are never invoked — that is, garbage collection, check-pointing, and task creation do not occur — in these measurements, because we measured the costs of creating and maintaining nested functions.

We measured the performance on a 1.05 GHz

Table 1 Performance measurements (for the creation and maintenance cost).

S: SPARC P: Pentium		Elapsed time in seconds (relative time to plain C)				
		C	GCC	LW-SC	XCC	CL-SC
BinTree copying GC	S	0.180 (1.00)	0.263 (1.46)	0.192 (1.07)	0.181 (1.00)	0.249 (1.38)
	P	0.152 (1.00)	0.169 (1.11)	0.156 (1.03)	0.150 (0.988)	0.179 (1.18)
Bin2List copying GC	S	0.292 (1.00)	0.326 (1.12)	0.303 (1.04)	0.289 (0.99)	0.318 (1.09)
	P	0.144 (1.00)	0.145 (1.01)	0.151 (1.05)	0.146 (1.01)	0.154 (1.07)
fib(34) check-pointing	S	0.220 (1.00)	0.795 (3.61)	0.300 (1.36)	0.226 (1.03)	0.361 (1.64)
	P	0.0628 (1.00)	0.152 (2.42)	0.138 (2.20)	0.0751 (1.20)	0.162 (2.58)
nqueens(13) load balancing	S	0.478 (1.00)	1.04 (2.18)	0.650 (1.36)	0.570 (1.19)	1.05 (2.20)
	P	0.319 (1.00)	0.428 (1.34)	0.486 (1.52)	0.472 (1.48)	0.544 (1.71)

```
(deftype sht (ptr (lightweight void void)))
(def (randinsert scan0 this n)
     (fn void sht (ptr Bintree) int)
  (decl i int)
  (decl k int)
  (decl seed (array unsigned-short 3))
  (def (scan1) (lightweight void void)
    (= this (move this))
    (scan0))
  (= (aref seed 0) 3)
  (= (aref seed 1) 4)
  (= (aref seed 2) 5)
  (for ((= i 0) (< i n) (inc i))
   (= k (nrand48 seed))
   (insert scan1 this k k)))
```

Fig. 14 The LW-SC program for BinTree.

```
(deftype sht (ptr (lightweight void void)))
(def (bin2list scan0 x rest)
     (fn (ptr Alist) sht (ptr Bintree) (ptr Alist))
  (def a (ptr Alist) 0)
  (def kv (ptr KVpair) 0)
  (def (scan1) (lightweight void void)
    (= x (move x))
    (= rest (move rest))
    (= a (move a))
    (= kv (move kv))
    (scan0))
  (if (fref (mref x) right)
      (= rest (bin2list scan1 (fref (mref x) right)
                        rest)) )
  (= kv (getmem scan1 (ptr KVpair_d)))
  (= (fref (mref kv) key) (fref (mref x) key))
  (= (fref (mref kv) val) (fref (mref x) val))
  (= a (getmem scan1 (ptr Alist_d)))
  (= (fref (mref a) kv) kv)
  (= (fref (mref a) cdr) rest)
  (= rest a)
  (if (fref (mref x) left)
      (= rest (bin2list scan1 (fref (mref x) left)
                        rest)))
  (return rest) )
```

Fig. 15 The LW-SC program for Bin2List.

UltraSPARC-III and a 3 GHz Pentium 4, using GCC with -O2 optimizers. **Table 1** summarizes the results of performance measurements,

```
(def (cpfib save0 n)
     (fn int (ptr (lightweight void)) int)
  (def pc int 0)
  (def s int 0)
  (def (save1) (lightweight void)
    (save0)
    (save-pc pc)
    (save-int s)
    (save-int n))
  (if (<= n 2)
      (return 1)
      (begin
       (= pc 1)
       (+= s (cpfib save1 (- n 1)))
       (= pc 2)
       (+= s (cpfib save1 (- n 2)))
       (return s))) )
```

Fig. 16 The LW-SC program for fib(34).

where "C" denotes a plain C program without high-level services, and "GCC" indicates the use of GCC's nested functions. "XCC" indicates the use of XC-cube, which is an extended C language with some primitives added for safe and efficient shared memory programming[23]. XC-cube also features nested functions with lightweight closures[21),22)], which are implemented at the assembly language level by modifying GCC directly☆. "CL-SC" (closure SC) indicates the use of nested functions with *non-lightweight* closures. Its implementation is almost the same as that of LW-SC except that all local variables and parameters are stored in the explicit stack.

Since nested functions are created frequently in fib(34), LW-SC performs well on the SPARC compared with GCC, where the cost of flushing instruction caches is significant. On the other

☆ The detail of its implementation will be reported by a separate paper[24].

Table 2 Performance measurements (for the invocation cost).

Elapsed time in seconds

		C	GCC	LW-SC	XCC	CL-SC
QSort (200,000)	SPARC	0.795	0.821	7.04	8.03	0.931
	(Ratio to C)	(1.00)	(1.03)	(8.86)	(10.1)	(1.17)
	Pentium	0.139	3.44	3.77	3.38	0.186
	(Ratio to C)	(1.00)	(24.7)	(27.1)	(24.3)	(1.33)
Bin2List copying GC	SPARC	—	0.495	0.522	0.495	0.526
	(GC time)		0.278	0.296	0.279	0.302
	Pentium	—	0.248	0.257	0.249	0.259
	(GC time)		0.0647	0.0685	0.0669	0.0714

```
(def (mod-sort a n d)
    (fn void (ptr int) int int)
  (def (comp-mod pp pq)
      (lightweight int (ptr void) (ptr void))
    (return
     (if-exp (< (% (mref (cast (ptr int) pp)) d)
                (% (mref (cast (ptr int) pq)) d))
             1
        (if-exp (== (% (mref (cast (ptr int) pp)) d)
                    (% (mref (cast (ptr int) pq)) d))
                0
                -1))))
  (quicksort a n (sizeof int) comp-mod))
```

Fig. 17 The LW-SC program of QSort (calling the sorting function by passing a nested function `comp-mod` as a comparator).

hand, LW-SC does not perform so well on the Pentium 4, where overhead with additional operations in LW-SC is emphasized.

Since several local variables can obtain callee-save registers in BinTree, LW-SC performs well on the SPARC, even if function calls (i.e., creations) are infrequent. This effect is not so significant in fib(34), since there are few local variable accesses in the `fib` function.

LW-SC does not perform well in nqueens(13), since unimportant variables are allocated to registers. Since the Pentium 4 has only a few callee-save registers and performs explicit save/restore of callee-save registers, which is implicit with the SPARC's register window, the penalty for wrong allocation is serious.

XC-cube performs better than LW-SC, mainly because it does not employ some of the additional operations in LW-SC, such as checking flags after returning from ordinary functions and at the beginning of function bodies (by using assembly-level techniques such as modifying return addresses). However, the difference is negligibly small if the body of a function is sufficiently large.

CL-SC performs worse than LW-SC, since all local variables and parameters are stored in the explicit stack and they never acquire registers.

6.2 Invocation Cost

To measure the cost of invoking nested functions, we employ the following programs:

QSort sorts 200,000 integers by using a quick sort algorithm invoking a nested function as a comparator, whose owner is the caller of the sorting function (**Fig. 17**). In the plain C program, the comparison function is defined as an ordinary function where `d` is declared as a global variable.

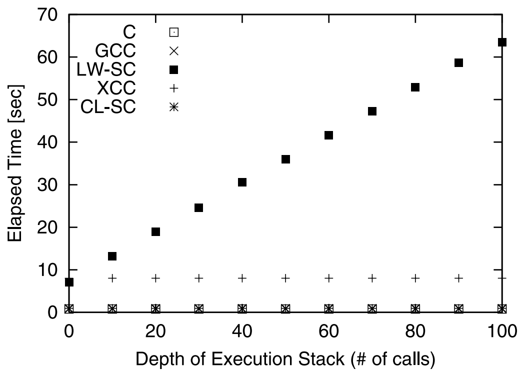
Bin2List (copying GC) works in the same way as Bin2List in Section 6.1, except that the garbage collector actually runs and nested functions are called for scanning the stack (and therefore there is no plain C program). The collectors employ a simple breadth-first (non-recursive) copying GC algorithm.

Table 2 summarizes the results of performance measurements. In LW-SC, the invocation cost is high because it is necessary to save (restore) the values in the execution stack upon calling (returning from) nested functions, which causes bad performance in QSort. What is worse, the cost of invoking a nested function increases according to the depth of the execution stack at the time of the invocation. To show this clearly, we invoked `mod-sort` in Fig. 17 on top of various numbers of intermediating function calls (**Fig. 18**). The results show that the elapsed time increases in proportion to the stack depth only in LW-SC. We think that the cost of throwing an exception to an exception handler may also change, for a similar reason.

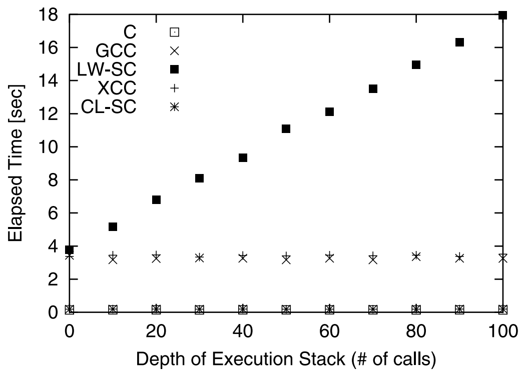
CL-SC performs well in QSort because unwinding and reconstruction of the execution stack are unnecessary.

Notice that GCC on Pentium performs badly in QSort. We guess that this is because trampoline code placed in a writable data area (not a read-only program area) prevents the processor from prefetching instructions.

All implementations show almost the same performance in Bin2List, even when only GC



UltraSPARC-III



Pentium 4

Fig. 18 Elapsed time in QSort against the number of intermediating function calls.

times are compared. This is because the invocation costs are negligible relative to the other costs for GC (such as scanning heaps).

These results show that LW-SC works effectively if nested functions are not called very frequently, and that CL-SC works better if they are called very often. Programmers and compiler writers can choose one of these implementations according to their situation.

7. Related Work

7.1 Compiler-Based Implementations of Nested Functions

As described above, GCC also features nested functions, but it is less portable and has high maintenance/creation costs. XC-cube implements nested functions with lightweight closures by modifying the GCC compiler. It performs better, but it also lacks portability.

7.2 Closure Objects in Modern Languages

Many modern languages such as Lisp and ML implement closures as first-class objects. These closure objects are valid after exit of their owner blocks. In most implementations they require some runtime support such as garbage collection, which makes C too inefficient to be used as an intermediate language to implement high-level languages.

7.3 Portable Assembly Languages

C−−[11),14)] also has an ability to access the variables sleeping in the execution stack by using the C−− runtime system to perform "stack walk." We expect that its efficiency is better than that of LW-SC, and almost equal to that of XC-cube. In terms of portability, LW-SC has the advantage that we can use pre-existing C compilers.

7.4 High-Level Services

This section lists high-level services, which are important applications of nested functions, and the techniques used for their implementation in previous work.

7.4.1 Garbage Collection

To implement garbage collection, the collector needs to be able to find all *roots*, each of which holds a reference to an object in the garbage-collected heap. In C, a caller's pointer variable may hold an object reference, but it may be sleeping in the execution stack until the return to the caller. Even when using direct stack manipulation, it is difficult for the collector to distinguished *roots* from other elements in the stack. For this reason, conservative collectors[1)] are normally used. Conservative copying collectors can inspect the execution stack but cannot modify it. Accurate copying of GC can be performed by using translation techniques based on "structure and pointer"[6),7)] with higher maintenance costs.

Figure 15 partially shows how scanning of *roots* can be implemented by using nested functions. `Getmem` allocates a new object in the heap and may invoke the copying collector with the nested function `scan1`. The copying collector can indirectly call `scan1`, which effects the movement (copying) of objects by using roots (`x`, `rest`, `a` and `kv`) and indirectly calls `scan0` in a nested manner. The actual entity of `scan0` may be another instance of `scan1` in the caller. The nested calls are performed until the bottom of the stack is reached.

7.4.2 Capturing/Restoring Stack State

Porch[16)] is a translator that transforms C programs into C programs supporting portable checkpoints. Portable checkpoints capture the state of a computation in a machine-independent format that allows the transfer of computations across binary incompatible machines. They introduce source-to-source compilation techniques for generating code to save and recover from such portable checkpoints automatically. To save the stack state, the program repeatedly returns and legitimately saves the parameters/local variables until the bottom of the stack is reached. During restoration, this process is reversed. Similar techniques can be used to implement migration and first-class continuations.

As shown in Fig. 16, the stack state can be captured without returning to the callers using nested functions. It uses techniques similar to those described above for scanning roots.

7.4.3 Multi-threads: Latency Hiding

Concert[13] and OPA[20] use similar translation techniques to support suspension and resumption of multiple threads on a single processor with a single execution stack (e.g., for latency hiding). They create a new child thread as an ordinary function call and, if the child thread completes its execution without being blocked, it simply returns the control to the parent thread. But in case of the suspension of the child thread, it legitimately saves its (live) parameters/local variables into heap-allocated frames and simply returns the control to the parent thread. When a suspended thread become runnable, it may legitimately restore necessary values from the heap-allocated frames.

The library implementation of StackThreads[19] provides two special service routines: `switch_to_parent` to save the context (state) of the child thread and transfer the control to the parent thread, and `restart_thread` to restore the context and transfer the control to the restarted thread. These routines are implemented in assembly languages by paying special attention to the treatment of callee-save registers.

StackThreads/MP[18] allows the frame pointer to walk the execution stack independently of the stack pointer. When the child thread is blocked, it can transfer the control to an arbitrary ancestor thread without copying the stack frames to the heap. StackThreads/MP employs the unmodified GNU C compiler and implements non-standard control-flows by using a combination of an assembly language postprocessor and runtime libraries.

Lazy Threads[5] employs a similar but different approach to frame management and thread suspension. Frames are allocated in a "stacklet," which is a small stack for several frames. A blocked child thread returns the control to the parent without copying the stack frame to the heap. When the parent is not at the top of the stacklet, it first allocates a new stacklet for allocating a stack frame. Lazy Threads is implemented by modifying the GNU C compiler.

Implementation techniques for multiple threads using nested functions are described in Refs. 17) and 8).

7.4.4 Load Balancing

To realize efficient dynamic load balancing by transferring tasks among computing resources in fine-grained parallel computing such as search problems, load balancing schemes which lazily create and extract a task by splitting the present running task, such as *Lazy Task Creation* (LTC)[12], are effective. In LTC, a newly created thread is directly and immediately executed like an ordinary call while (the *continuation* of) the oldest thread in the computing resource may be stolen by other idle computing resources. Usually, the idle computing resource (*thief*) randomly selects another computing resource (*victim*) from which to steal a task.

Compilers (translators) for multithreaded languages generate low-level code. In the original LTC[12], assembly code is generated to directly manipulate the execution stack. Translators for both Cilk[4] and OPA[20] generate C code. Since it is illegal and not portable for C code to directly access the execution stack, the Cilk and OPA translators generate two versions (fast/slow) of code; the fast version code saves the values of live variables in a heap-allocated frame upon call (in the case of Cilk) or return (in the case of OPA) so that the slow version code can continue the rest of computation based on the heap-allocated saved *continuation*.

A message-passing implementation[3] of LTC employs a polling method where the *victim* detects a task request sent by the *thief* and returns a new task created by splitting the present running task. This technique enables OPA[20], StackThreads/MP[18], and Lazy Threads[5] to support load balancing.

We restructured LTC with backtracking, where callers' variable are accessed by using nested functions for infrequent task creation[21],[22].

8. Conclusion and Future Work

This paper has presented a technique for implementing nested functions for the C language, employing the SC language system. Since the implementation is transformation-based, it allows implement high-level services with "stack walk" to be implemented in a portable way. Furthermore, such services can be implemented efficiently because we aggressively reduce the cost of creating and maintaining nested functions by using "lightweight" closures. Future work includes actual implementation of high-level languages with such services (e.g., providing a garbage-collected heap with a copying collector).

Acknowledgments This work was partly supported by the Japan Society for the Promotion of Science (JSPS), the 21st Century COE Program (Grant No.14213201), and a MEXT Grant-in-Aid for Exploratory Research (17650008).

References

1) Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice & Experience*, Vol.18, No.9, pp.807–820 (1988).
2) Breuel, T.M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).
3) Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proc. International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, No.748, Springer-Verlag, pp.94–107 (1993).
4) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices* (*PLDI '98*), Vol.33, No.5, pp.212–223 (1998).
5) Goldstein, S.C., Schauser, K.E. and Culler, D.E.: Lazy Threads: Implementing a Fast Parallel Call, *Journal of Parallel and Distributed Computing*, Vol.3, No.1, pp.5–20 (1996).
6) Hanson, D.R. and Raghavachari, M.: A Machine-Independent Debugger, *Software — Practice & Experience*, Vol.26, No.11, pp.1277–1299 (1996).
7) Henderson, F.: Accurate Garbage Collection in an Uncooperative Environment, *Proc. 3rd International Symposium on Memory Management*, pp.150–156 (2002).
8) Hiraishi, T., Li, X., Yasugi, M., Umatani, S. and Yuasa, T.: Language Extension by Rule-Based Transformation for S-Expression-Based C Languages, *IPSJ Transactions on Programming*, Vol.46, No.SIG 1(PRO 24), pp.40–56 (2005). (in Japanese).
9) Hiraishi, T., Yasugi, M. and Yuasa, T.: Effective Utilization of Existing C Header Files in Other Languages with Different Syntaxes, *7th Workshop on Programming and Programming Languages* (*PPL2005*) (2005). (in Japanese).
10) Hiraishi, T., Yasugi, M. and Yuasa, T.: Implementing S-Expression Based Extended Languages in Lisp, *Proceedings of the International Lisp Conference*, Stanford, CA, pp.179–188 (2005).
11) Jones, S.P., Ramsey, N. and Reig, F.: C−−: A Portable Assembly Language That Supports Garbage Collection, *International Conference on Principles and Practice of Declarative Programming* (1999).
12) Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (1991).
13) Plevyak, J., Karamcheti, V., Zhang, X. and Chien, A.A.: A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers, *Supercomputing'95* (1995).
14) Ramsey, N. and Jones, S.P.: A Single Intermediate Language That Supports Multiple Implementations of Exceptions, *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pp.285–298 (2000).
15) Stallman, R.M.: Using and Porting GNU Compiler Collection (1999).
16) Strumpen, V.: Compiler Technology for Portable Checkpoints, `http://theory.lcs.mit.edu/~porch/` (1998).
17) Tabata, Y., Yasugi, M., Komiya, T. and Yuasa, T.: Implementation of Multiple Threads by Using Nested Functions, *IPSJ Transactions on Programming*, Vol.43, No.SIG 3(PRO 14), pp.26–40 (2002). (in Japanese).
18) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPoPP*), pp.60–71 (1999).
19) Taura, K. and Yonezawa, A.: Fine-Grain Multithreading with Minimal Compiler Support: A Cost Effective Approach to Implementing Efficient Multithreading Languages, *Proc. Conference on Programming Language Design and Implementation*, pp.320–333 (1997).
20) Umatani, S., Yasugi, M., Komiya, T. and Yuasa, T.: Pursuing Laziness for Efficient Implementation of Modern Multithreaded Lan-

guages, *Proc. 5th International Symposium on High Performance Computing*, Lecture Notes in Computer Science, No.2858, pp.174–188 (2003).

21) Yasugi, M., Komiya, T. and Yuasa, T.: Dynamic Load Balancing by Using Nested Functions and Its High-Level Description, *IPSJ Transactions on Advanced Computing Systems*, Vol.45, No.SIG 11(ACS 7), pp.368–377 (2004). (in Japanese).

22) Yasugi, M., Komiya, T. and Yuasa, T.: An Efficient Load-Balancing Framework Based on Lazy Partitioning of Sequential Programs, *Proc. Workshop on New Approaches to Software Construction*, pp.65–84 (2004).

23) Yasugi, M., Takada, J., Tabata, Y., Komiya, T. and Yuasa, T.: Primitives for Shared Memory and Its Implementation with GCC, *IPSJ Transactions on Programming*, Vol.43, No.SIG 1(PRO 13), pp.118–132 (2002). (in Japanese).

24) Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *15th International Conference on Compiler Construction* (2006). (to appear).

Appendix

An example of translation from LW-SC into SC-0

```
;;; The pointer to the moved "nested function".
(deftype nestfn-t
        (ptr (fn (ptr char) (ptr char) (ptr void))))
;;; The structure which contains the pointer to the moved
;;; nested function and the frame pointer of
;;; the owner function.
(deftype closure-t struct
  (def fun nestfn-t)
  (def fr (ptr void)))

(deftype align-t double)

;;; The auxiliary function for calling nested functions.
(def (lw-call esp) (fn (ptr char) (ptr char))
  (def clos (ptr closure-t)
           (mref (cast (ptr (ptr closure-t)) esp)))
  (return ((fref clos -> fun) esp (fref clos -> fr))))

;;; The frame structure of function h.
(def (struct h_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def tmp2 int)
  (def tmp int)
  (def g (ptr closure-t))
  (def i int))

(def (h esp i g)
    (fn int (ptr char) int (ptr closure-t))
  (def argp (ptr char))
  (def efp (ptr (struct h_frame)))
  (def new-esp (ptr char))
  (def esp-flag size-t (bit-and (cast size-t esp) 3))
  (def tmp int)
  (def tmp2 int)
  (def tmp_fp (ptr closure-t))
  (def tmp_fp2 (ptr closure-t))

  ;; Judge whether reconstruction of the execution stack is
  ;; required or not.
  (if esp-flag
      (begin
        (= esp (cast (ptr char)
                (bit-xor (cast size-t esp) esp-flag)))
        (= efp (cast (ptr (struct h_frame)) esp))
        ;; Move the stack pointer by the frame size.
        (= esp
           (cast (ptr char)
                 (+ (cast (ptr align-t) esp)
                    (/ (+ (sizeof (struct h_frame))
                          (sizeof align-t) -1)
                       (sizeof align-t)))))
        (= (mref (cast (ptr (ptr char)) esp)) 0)
        ;; Restore the execution point.
        (label LGOTO
          (switch (fref (mref efp) call-id)
            (case 0) (goto l_CALL)
            (case 1) (goto l_CALL2)))
        (goto l_CALL)))
  (= efp (cast (ptr (struct h_frame)) esp))
  ;; Move the stack pointer by the frame size.
  (= esp
     (cast (ptr char)
           (+ (cast (ptr align-t) esp)
              (/ (+ (sizeof (struct h_frame))
                    (sizeof align-t) -1)
                 (sizeof align-t)))))
  (= (mref (cast (ptr (ptr char)) esp)) 0)
  ;; Call the nested function g.
  (begin
    (= tmp_fp g)
    (= argp
       (cast (ptr char)
             (+ (cast (ptr align-t) esp)
                (/ (+ (sizeof (ptr char))
                      (sizeof align-t) -1)
                   (sizeof align-t)))))
    ;; Push the arguments passed to nested function.
    (exps (= (mref (cast (ptr int) argp)) i)
          (= argp
             (cast (ptr char)
               (+ (cast (ptr align-t) argp)
                  (/ (+ (sizeof int)
                        (sizeof align-t) -1)
                     (sizeof align-t))))))
    ;; Push the structure object that corresponds to
    ;; the frame of the nested function to
    ;; the explicit stack.
    (= (mref (cast (ptr (ptr closure-t)) argp)) tmp_fp)
    ;; Save the values of local variables to the frame.
    (= (fref efp -> tmp2) tmp2)
    (= (fref efp -> tmp) tmp)
    (= (fref efp -> g) g)
    (= (fref efp -> i) i)
    (= (fref efp -> argp) argp)
    (= (fref efp -> tmp-esp) argp)
    ;; Save the current execution point.
    (= (fref efp -> call-id) 0)
    (return (- (cast int 0) 1))
    ;; Continue the execution from here after the function call finishes.
    (label l_CALL nil)
    ;; Restore local variables from the explicit stack.
    (= tmp2 (fref efp -> tmp2))
    (= tmp (fref efp -> tmp))
    (= g (fref efp -> g))
    (= i (fref efp -> i))
    ;; Get the return value.
    (= tmp (mref (cast (ptr int) (fref efp -> argp)))))
  ;; Call the nested function g.
  (begin
    (= tmp_fp2 g)
    (= argp
       (cast (ptr char)
          (+ (cast (ptr align-t) esp)
             (/ (+ (sizeof (ptr char))
                   (sizeof align-t) -1)
                (sizeof align-t)))))
    ;; Push the arguments passed to nested function.
    (exps (= (mref (cast (ptr int) argp)) tmp)
          (= argp
             (cast (ptr char)
                (+ (cast (ptr align-t) argp)
                   (/ (+ (sizeof int)
                         (sizeof align-t) -1)
                      (sizeof align-t))))))
    ;; Push the structure object that corresponds to
    ;; the frame of the nested function to
    ;; the explicit stack.
    (= (mref (cast (ptr (ptr closure-t)) argp))
       tmp_fp2)
    ;; Save the values of local variables to the frame.
    (= (fref efp -> tmp2) tmp2)
    (= (fref efp -> tmp) tmp)
    (= (fref efp -> g) g)
    (= (fref efp -> i) i)
    (= (fref efp -> argp) argp)
    (= (fref efp -> tmp-esp) argp)
    ;; Save the current execution point.
    (= (fref efp -> call-id) 1)
    (return (- (cast int 0) 1))
    ;; Continue the execution from here after
    ;; the function call finishes.
    (label l_CALL2 nil)
    (= tmp2 (fref efp -> tmp2))
    (= tmp (fref efp -> tmp))
    (= g (fref efp -> g))
    (= i (fref efp -> i))
    ;; Get the return value.
    (= tmp2 (mref (cast (ptr int)
                        (fref efp -> argp)))))
  (return tmp2))

;;; The frame structure of function foo.
(def (struct foo_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def tmp3 int)
  (def y int)
  (def x int)
  (def a int)
  (def g10 closure-t))

;;; The frame structure of function g1 .
(def (struct g1_in_foo_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def b int)
  (def xfp (ptr (struct foo_frame))))

;;; Nested function g1 (moved to the top-level).
(def (g1_in_foo esp xfp0)
     (fn (ptr char) (ptr char) (ptr void))
 (def new-esp (ptr char))
 (def efp (ptr (struct g1_in_foo_frame)))
 ;; The frame pointer of the owner function.
 (def xfp (ptr (struct foo_frame)) xfp0)
 (def esp-flag size-t (bit-and (cast size-t esp) 3))
 (def parmp (ptr char)
   (cast (ptr char)
         (bit-xor (cast size-t esp) esp-flag)))
 ;; Pop parameters from the explicit stack.
 (def b int
   (exps
     (= parmp
        (cast (ptr char)
              (- (cast (ptr align-t) parmp)
                 (/ (+ (sizeof int) (sizeof align-t) -1)
                    (sizeof align-t)))))
     (mref (cast (ptr int) parmp))))

 (label LGOTO nil)
 (= efp (cast (ptr (struct g1_in_foo_frame)) esp))
 ;; Move the stack pointer by the frame size.
 (= esp
    (cast (ptr char)
          (+ (cast (ptr align-t) esp)
             (/ (+ (sizeof (struct g1_in_foo_frame))
                   (sizeof align-t) -1)
                (sizeof align-t)))))
 (= (mref (cast (ptr (ptr char)) esp)) 0)
 (inc (fref xfp -> x))
 ;; Push the return value to the explicit stack.
 (= (mref (cast (ptr int) efp)) (+ (fref xfp -> a) b))
 (return 0))

(def (foo esp a) (fn int (ptr char) int)
  (def efp (ptr (struct foo_frame)))
  (def new-esp (ptr char))
  (def esp-flag size-t (bit-and (cast size-t esp) 3))
  (def x int 0)
  (def y int 0)
  (def tmp3 int)

  ;; Judge whether reconstruction of the execution stack is
  ;; required or not.
  (if esp-flag
      (begin
        (= esp (cast (ptr char)
                     (bit-xor (cast size-t esp) esp-flag)))
        (= efp (cast (ptr (struct foo_frame)) esp))
        ;; Move the stack pointer by the frame size.
        (= esp
           (cast (ptr char)
                 (+ (cast (ptr align-t) esp)
                    (/ (+ (sizeof (struct foo_frame))
                          (sizeof align-t) -1)
                       (sizeof align-t)))))
        (= (mref (cast (ptr (ptr char)) esp)) 0)
        (label LGOTO
          ;; Restore the execution point.
          (switch (fref (mref efp) call-id)
            (case 0) (goto l_CALL3)))
        (goto l_CALL3)))
  (= efp (cast (ptr (struct foo_frame)) esp))
  ;; Move the stack pointer by the frame size.
  (= esp
     (cast (ptr char)
           (+ (cast (ptr align-t) esp)
              (/ (+ (sizeof (struct foo_frame))
                    (sizeof align-t) -1)
                 (sizeof align-t)))))
  (= (mref (cast (ptr (ptr char)) esp)) 0)
  (= new-esp esp)
  ;; Call the ordinary function h.
  (while
      (and
       (== (= tmp3 (h new-esp 10
                      (ptr (fref
                       (cast (ptr (struct foo_frame))
                             esp)
                      -> g10))))
           (- (cast int 0) 1))
       (!= (= (fref efp -> tmp-esp)
              (mref (cast (ptr (ptr char)) esp))) 0))
    ;; Save the values of local variables to the frame.
    (= (fref efp -> tmp3) tmp3)
    (= (fref efp -> y) y)
    (= (fref efp -> x) x)
    (= (fref efp -> a) a)
    (= (fref efp -> g10 fun) g1_in_foo)
    (= (fref efp -> g10 fr) (cast (ptr void) efp))
    ;; Save the current execution point.
    (= (fref efp -> call-id) 0)
    (return (- (cast int 0) 1))
    ;; Continue the execution from here after
    ;; the function call finishes.
    (label l_CALL3 nil)
    ;; Restore local variables from the explicit stack.
    (= tmp3 (fref efp -> tmp3))
    (= y (fref efp -> y))
    (= x (fref efp -> x))
    (= a (fref efp -> a))
    (= new-esp (+ esp 1)))
  (= y tmp3)
  (return (+ x y)))

;;; The frame structure of function main .
(def (struct main_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def tmp4 int))

(def (main) (fn int)
  (def efp (ptr (struct main_frame)))
  (def new-esp (ptr char))
  (def estack (array char 65536))  ; The explicit stack.
  (def esp (ptr char) estack)
  (def tmp4 int)

  (label LGOTO nil)
  (= efp (cast (ptr (struct main_frame)) esp))
  ;; Move the stack pointer by the frame size.
  (= esp
     (cast (ptr char)
           (+ (cast (ptr align-t) esp)
              (/ (+ (sizeof (struct main_frame))
                    (sizeof align-t) -1)
                 (sizeof align-t)))))
  (= (mref (cast (ptr (ptr char)) esp)) 0)
  (= new-esp esp)
  (while
      (and
       (== (= tmp4 (foo new-esp 1))
           (- (cast int 0) 1))
       (!= (= (fref efp -> tmp-esp)
              (mref (cast (ptr (ptr char)) esp))) 0))
    (def goto-fr (ptr char))
    (= (mref (cast (ptr (ptr char)) esp)) 0)
    (= (fref efp -> tmp4) tmp4)
    ;; Execute nested functions.
    (= goto-fr (lw-call (fref efp -> tmp-esp)))
    (if (== (cast (ptr char) goto-fr)
            (cast (ptr char) efp))
        (goto LGOTO))
    (= new-esp (+ esp 1)))
  (return tmp4))
```

(Received September 22, 2005)
(Accepted January 11, 2006)

Tasuku Hiraishi was born in 1981. He received the B.E. degree in Information Science in 2003, the Master of Informatics in 2005, from Kyoto University. He is currently a Ph.D. candidate at Graduate School of Informatics, Kyoto University. His research interests include programming languages and parallel processing. He is a student member of the Japan Society for Software Science and Technology.

Masahiro Yasugi was born in 1967. He received a B.E. in electronic engineering, an M.E. in electrical engineering, and a Ph.D. in information science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. Since 1998, he has been working at Kyoto University as an assistant professor (lecturer) and an associate professor (since 2003). In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of the ACM and the Japan Society for Software Science and Technology.

Taiichi Yuasa received a Bachelor of Mathematics degree in 1977, the Master of Mathematical Sciences degree in 1979, and a Doctor of Science degree in 1987, all from Kyoto University, Kyoto, Japan. He joined the faculty of the Research Institute for Mathematical Sciences, Kyoto University, in 1982. He is currently a Professor at the Graduate School of Informatics, Kyoto University, Kyoto, Japan. His current areas of interest include symbolic computation and programming language systems. He is a member of the ACM, the IEEE, the Institute of Electronics, Information and Communication Engineers, and the Japan Society for Software Science and Technology.