Implementing S-Expression Based Extended Languages in Lisp

Tasuku Hiraishi Masahiro Yasugi Taiichi Yuasa

{hiraisi, yasugi, yuasa}@kuis.kyoto-u.ac.jp
Graduate School of Informatics, Kyoto University
Sakyo Kyoto, JAPAN 606-8501

ABSTRACT

Many extended, C-like languages can be implemented by translating them into C. This paper proposes an extension scheme for SC languages (extended/plain C languages with an S-expression based syntax). The extensions are implemented by transformation rules over S-expressions, that is, Lisp functions with pattern-matching on S-expressions. Thus, many flexible extensions to C can be implemented at low cost because (1) of the ease with which new constructs can be added to an SC language, and (2) of the pre-existing Common Lisp capabilities for reading/printing, analyzing, and transforming S-expressions themselves. We also present a practical example of just such an extended language.

Keywords

C, Lisp, language extensions, nested functions, intermediate languages

1. INTRODUCTION

The C language is often indispensable for developing practical systems, but it is not so easy to extend the C language by adding a new feature such as fine-grain multi-threading. We can implement language extension by modifying a C compiler, but sometimes we can do it by translating an extended C program to an Abstract Syntax Tree (AST), apply analysis or transformation necessary for the extension, and then generate C code. Structures, objects (in object-oriented languages), or variants are traditionally used as the data structure for an AST.

This paper proposes a new scheme where an AST is represented by an S-expression and such an S-expression is also used as (a part of) a program. For this purpose we have designed SC, a C language with an S-expression-based syntax. This scheme enables us to implement language extension at low cost because (1) adding new constructs is easy, (2) S-expressions can easily be read/printed, analyzed, and transformed in Common Lisp, which features dynamic variables useful for transformation. We also developed the SC language system in Common Lisp. In this language system, extension developers can implement their extensions by writing some transformation rules over S-expressions. The rules are described as function definitions with pattern-matching on their arguments.

We have implemented some language extensions such as multi-threading and check-pointing using this system, and this paper mainly shows one of such language extensions, LW-SC (Lightweight-SC), where nested functions are added to SC-0 as a language feature.

This system is helpful especially for programming language developers who want to prototype their implementation ideas rapidly and also useful for C programmers who want to customize the language easily as Lisp programmers usually do.

2. THE SC LANGUAGE SYSTEM

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 kinds of modules:

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

Figure 1 shows code translation phases in the SC language system. An extended SC code is transformed into SC-0 by the SC translators, then translated into C by the SC compiler. Before each transformation/translation is applied, preprocessing by an SC preprocessor is performed. As the figure shows, a series of rule-sets can be applied one by one to get SC-0 code. Extension implementers write transformation rules for the SC translators to transform the extended language into SC-0.

2.1 The SC Preprocessor

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

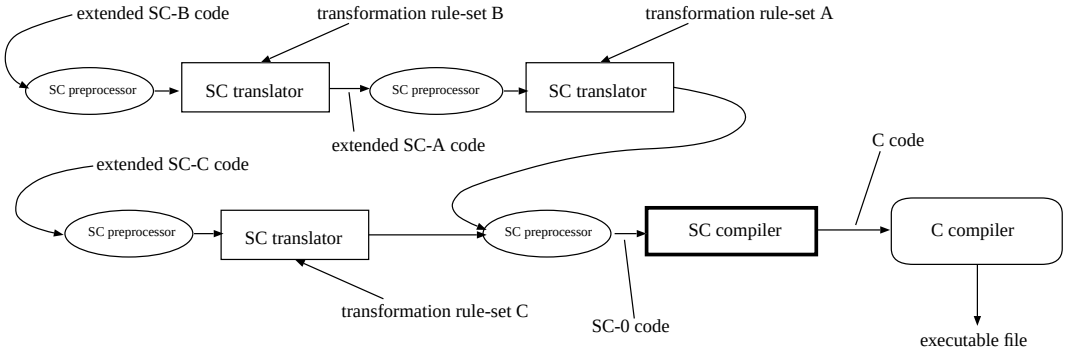

Figure 1: Code translation phases in the SC language system.

- (`%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 `eq`s *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, then the result is evaluated by Common Lisp. If the return value `eq`ls `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. The SC programmers can use library functions and most of macros such as `printf`, `NULL` declared/`#defined` in C header files[1].

2.2 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 usual Lisp function. When the function is called, the first argument is tested whether it matches to *pattern*. If matched, *expression* is evaluated by the Common Lisp system, then its value is returned as the result of the function call. The parameters *parm*$_2 \cdots$ *parm*$_n$, if any, are treated as usual arguments.

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

It is permitted to abbreviate

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

(all the *expression*s are identical and only *pattern*s are different from each other) to

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

The *pattern* is an S-expression consisted 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*`]`

[1] 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 discussed this problem before in [1].

matches an *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 above or an usual Common Lisp function (a built-in function or what is defined separately from transformation rules).

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

An example of such a function definition is as follows[2]:

```
(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 usual Common Lisp function definition (`defun`). The output can be loaded by the `load` function, which enables the programmers to easily test a part of a transformation rule-sets in an interactive environment.

2.3 The SC Compiler and The SC-0 Language

We designed the SC-0 language as the final target language of transformation by transformation rules. It has the following features:

- an S-expression based, Lisp like syntax,
- the C semantics; actually most of C code can be represented in SC-0 in a straightforward manner[3], and
- practical for programming.

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

[2]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 inconvenience that programmers have to put backquotes before most of patterns. We preferred shorter descriptions and adopt the notation without backquotes.

[3]except some features such as `->` operators, `for` constructs, and `while` constructs. These are implemented as language extensions of the SC-0 using the SC language system itself.

```
(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))
```

Figure 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;
}
```

Figure 3: An equivalent C program to Figure 2.

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

3. AN EXAMPLE OF A LANGUAGE EXTENSION — LW-SC

This section presents an example of an extended language using the SC language system, named LW-SC. In LW-SC, nested functions are added to SC-0 as a language feature. That is, programmers are permitted to write function definitions within another function.

This extension is not only shown as an introduction of the SC language system, but practical itself; those nested functions can access a caller's local variables directly without returning from the callee, that enables us to implement many high-level services such as check-pointing, multi-threading[2, 3] and garbage collection easily and elegantly by using LW-SC as an intermediate language.

The GNU C Compiler[4] (GCC) also provides such nested functions as an extension to C. But LW-SC is more portable because it is implemented by code transformation to C, while GCC's nested functions are implemented as an extended C compiler. Moreover, using nested functions of GCC causes high overhead for the allocation and maintenance of them. We have overcome this problem by implementing nested functions with "lightweight" closures (with some insignificant restrictions). Lightweight closures causes higher invocation overhead but they have quite little allocation/maintenance overhead. As far as nested functions are basically used for high-level services described above, the total overhead can be reduced significantly since most such services allocate closures frequently but only call them infrequently (e.g., to scan roots in garbage collection). We detail the performance of LW-SC in Section 3.4.

3.1 Language Specification

LW-SC has the following features added 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 per-

```
(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)))
```

Figure 4: An LW-SC program.

mitted to be the nested function pointer type other than the ordinary function pointer type.

- **Defining nested functions:** In the 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 almost same syntax as ordinary function definitions' except for the difference between keywords `fn` and `lightweight`.)

A nested function can access the lexically-scoped variables in the allocation-time environment and its pointer can be used as a function pointer to indirectly call the closure. For example, Figure 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.

As well as GCC (but differently from Lisp's closure objects), nested functions are valid only when the owner blocks are alive. Unlike GCC, pointers to nested functions are not compatible with ones to top-level functions. However, such limitations are insignificant for the purpose of implementing high-level services mentioned above.

3.2 Transformation Strategy

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

3.2.1 Basic Ideas

The basic ideas to implement nested functions by translation are summarized as follows:

- After transformation, all definitions of nested functions are 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 maintenance/allocation overhead, 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 based on these ideas.

3.2.2 Transformation

LW-SC programs are translated in the following way to realize the ideas described in Section 3.2.1.

(a) Each generated C program employs an explicit stack mentioned above on 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 to the same function call in C, except that a special argument is added which saves the stack pointer on the explicit stack. The callee firstly initializes its frame pointer with the stack pointer, moves the stack pointer by its frame size, then executes its body.

(c) Each nested function definition in LW-SC is moved to the top-level in C. In the original place, a variable of a structure type, which contains the pointer to the moved nested function and the frame pointer of the owner function, is declared instead.

(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 form the function.
3. Iterate Step 2 until returned to `main`. The values of local variables and parameters of `main` are also stored to 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 firstly 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 referring to the values saved in the explicit stack at Step 3 (the values may be changed during the call to the nested function) to return to 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.

A callee (nested functions) 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, Figure 5 shows the state transition of the two stacks[4], in the case of Figure 4, from the beginning of the execution to the end of the first indirect call to a nested function `g1` (Each number

[4] "The C stack" here just states the set of local variables and parameters, whose values are stored not only in the stack memory but also in registers.

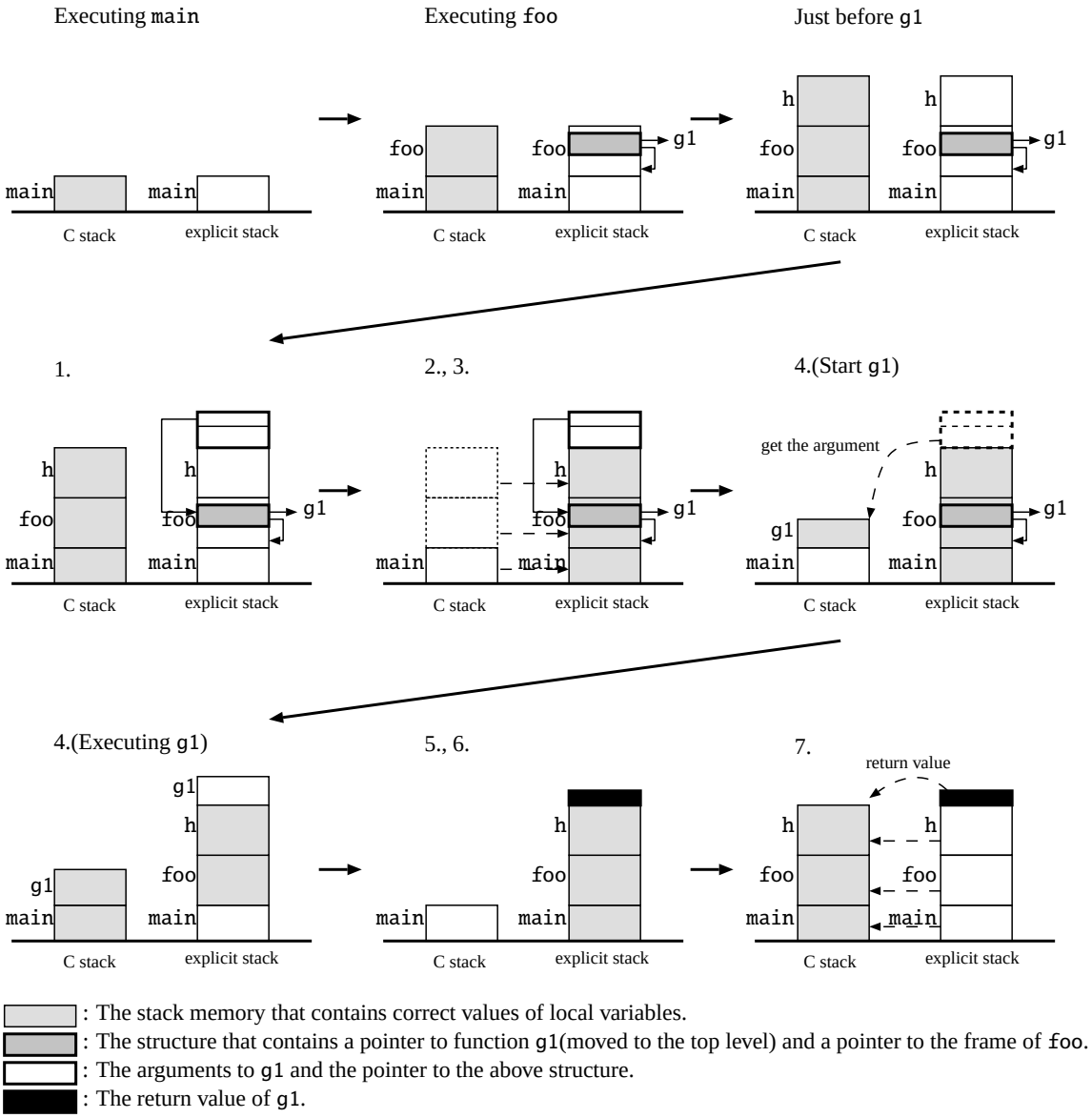

Figure 5: Details of an indirect call to a nested function `g1` in Figure 4.

in the figure corresponds to the step of the nested function call described at (d)). Notice that the correct values of the local variables are saved in the explicit stack during the execution of the nested function and otherwise in the C stack.

3.3 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 simpleness 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 3.2.2.

(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 detail of these transformation rule-sets.

3.3.1 The `type` *rule-set*

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

Figure 6 shows the (abbreviated) transformation rule-set. `Tp0` is

applied to input program (e.g., in Figure 7) to get the transformed program (e.g., in Figure 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.

3.3.2 *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 makes such function calls not appear.

Because some temporary variables are needed for the transformation, the definitions of those are inserted at the head of the function body. For example, a program in Figure 10 is transformed to the program in Figure 11 using this rule-set.

Figure 9 shows the (abbreviated) `temp` rule-set. The actual transformations are 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.

3.3.3 *The* `lightweight` *rule-set*

Now the transformation described in Section 3.2.2 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. `esp` appearing in the code is a special parameter which is added to each function and saves the stack pointer of the explicit stack. `efp` is a special local variable added to each function, which saves the frame pointer of the function. `Lwe-xfp` transforms references to local variables into references to the explicit stack.

"Ordinary function" calls and "nested function" calls can be statically distinguished referring to the functions' type because of the restriction that ordinary function types are incompatible with nested function types.

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

Calling ordinary functions The function call is performed as a part of the condition expression of `while`, where the stack pointer is passed to the callee as an additional first argument. If the callee procedure normally finished, the condition becomes false and the body of `while` loop is not executed. Otherwise, if the callee returned because of a "nested function" call, the condition becomes true. In the body of `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 (`efp->called-id`),

```
(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 "sytax error")
;;;;;;; body ;;;;;;
(Tpb (do-while ,exp ,@body))
-> (switch ,(Tpe exp)
     ,@(let ((*v-alist* *v-alist*)
             (*str-alist* *str-alsit*))
         (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
```

Figure 6: The `type` rule-set (abbreviated).

```
;;;; Due to the temp rule-set, a function call expression must be appeared in either of the following form 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 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 objcect 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))) )) )))
```

Figure 12: The `lightweight` rule-set (abbriviated).

and then the current function temporarily exits. This function is re-called when reconstructing the execution stack after the execution of the nested function. Then the control is transferred to the `label` that is put next to the `return` by `goto` statements which are added in the head of the function. Then the values of local variables are restored form the explicit stack and the function in the condition expression of `while` is re-called. The assignment `(= new-esp (+ esp 1))` at the end of `while` block set a flag at the LSB of the explicit stack pointer that indicates reconstructing the execution stack.

Calling Nested functions The arguments passed to the nested function and the structure that contains the nested function pointer and the frame pointer of its owner function are pushed to the explicit stack. Then, like 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 procedure for calling nested function. Then the values of local variables are restored and the return value of the nested function is taken from the top of the explicit stack, if necessary.

Returning from functions `return`s from ordinary function need no transformation. On the other hand, `return`s from nested functions must be transformed to push the return value to the explicit stack, and just to `return` 0 to indicate that the execution of the function is normally finished.

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

- initializing the frame pointer of the explicit stack (`efp`) and the stack pointer (`esp`),
- judging whether reconstruction of the execution stack is required or not and, if so, executing `goto` to the `label`

```
(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))))
```

Figure 7: An example transformation by the `type` rule-set (before).

```
(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)))))))
```

Figure 8: An example transformation by the `type` rule-set (after).

corresponding to `efp->call-id`, and

- popping parameters from the explicit stack, in the case of nested functions.

Other transformations are also needed such as adding the parameter `esp` that receives the explicit stack pointer, adding some local variable definitions, and adding the structure definition that represents the function's frame in the explicit stack and is referred to by `efp`.

Though more transformations are needed, we omit the details due to the space limitation.

3.3.4 The `untype` *rule-set*

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

3.4 Performance

We employed several programs with nested functions and compared them with the corresponding plain C programs. We measured the performance on 1.05GHz UltraSPARC-III and 3GHz Pentium 4 using GCC with `-O2` optimizers. Table 1 summarizes the results of performance measurements, where "C" means the plain C program, and "GCC" means the use of GCC's nested functions.

GCC's implementation of nested functions causes high maintenance/allocation overhead for the following reasons:

- GCC implements taking the address of a nested function using a technique called *trampolines*[5]. Trampolines are code fragments generated on the stack at runtime to indirectly enter the nested function with a necessary environment. The

```
(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)))))
...
```

Figure 9: The `temp` rule-set (abbriviated).

```
(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)))))))
```

Figure 10: An example transformation by the `temp` rule-set (before).

```
(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)))))
```

Figure 11: An example transformation by the `temp` rule-set (after).

```
(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
```

Figure 13: The `untype` rule-set.

runtime code generation incurs high overhead, and for some processors like SPARC, it is necessary to flush some instruction caches for the runtime-generated trampoline code.

- The local variables generally may get registers if the owner function has no nested function. But an owner of GCC's nested functions keeps the values of these variables in the stack for the nested functions to access them usually via a static chain. Thus, the owner function must perform memory operations to access these variables, which incurs high maintenance overhead.

LW-SC overcomes the former problem by translating the nested function pointer to the tuple of the ordinary function pointer and the frame pointer, and the latter by saving the local variables to the explicit stack lazily (only on call to nested functions).

Actually LW-SC shows a good performance on SPARC because overhead for flushing instruction caches is significant. On the other hand, LW-SC does not show good performance on Pentium 4. In fib(36), overhead for additional operations in LW-SC is emphasized since there is little local variable access in the `fib` function.

Table 1: Performance Measurements.

S:SPARC P:Pentium		Elapsed Time in seconds (relative time to plain C) C	GCC	LW-SC
BinTree copying GC	S	0.251 (1.00)	0.335 (1.33)	0.274 (1.09)
	P	0.149 (1.00)	0.170 (1.14)	0.152 (1.02)
Bin2List copying GC	S	0.415 (1.00)	0.467 (1.13)	0.423 (1.02)
	P	0.144 (1.00)	0.145 (1.01)	0.151 (1.05)
fib(36) Check Pointing	S	0.341 (1.00)	1.518 (4.45)	0.412 (1.21)
	P	0.0702 (1.00)	0.114 (1.62)	0.146 (2.08)

Table 2: The number of lines of each transformation rule-set.

	Lines
The `type` rule-set	450
The `temp` rule-set	340
The `lightweight` rule-set	780
The `untype` rule-set	10

3.5 Implementation Cost

Table 2 shows the number of lines of each transformation rule-set. Because a program to be transformed is given as S-expressions, the transformation rules can be written intuitively and easily. It is easy to test transformation rules, too. For example, an input for the `temp` rule-set can be written in a simple S-expression and the output is also easily checked. In addition, the `type`, `temp` and `untype` rule-set can often be reused when implementing some other extensions.

3.6 Debugging SC Programs

When SC programmers debug their SC programs, they may read the generated C code but it is usually too complicated. We will solve this problem by making transformation rules weave debugging code into their output.

4. RELATED WORK

4.1 Other Language Extensions

There exists many useful extensions to C such as Cilk[6] and OpenMP[7], but their purpose is to implement their own extensions, not to make a framework for general language extensions.

Lisp/Scheme is easy to be extended by transformation and to be written by humans. Furthermore, there exists compilers from their languages to C[8, 9]. But they are not suitable for describing low level operations such as pointer operations. From another point of view, the SC language system applies the advantages of Lisp in extensibility to C with preservation of its ability of low-level operations.

4.2 Lower-Level Scheme

Pre-Scheme[10] is a dialect of Scheme, which lacks some features such as garbage collection and full proper tail-recursion but preserves the other Scheme's advantages such as higher order procedures, interactive debugging, and macros and gives low-level machine access of C. Our approach is to support language develop-

ers to implement language extension rather than to support programmers for low-level programming, using some advantages of Lisp/Scheme.

4.3 Reflection

Reflection is to manipulate behaviors of a running program by referring to or modifying meta-level information as a first-class object, which enables us to extend a program dynamically. Although it is very powerful in extensibility, it causes great decrease of performance in many implementations that such information is kept by an interpreter. Most implementation[11] overcomes this problem by restricting the extension targets. *Compile-time reflection*[12, 13] realizes such extension by transforming programs in compile-time, which is similar to our approach. But we provide more generic framework to transform programs such as from LW-SC into SC-0.

4.4 Aspect Oriented Programming

Aspect Oriented Programming[14] (AOP) is a programming paradigm to handle a cross-cutting concern in one place. In general, it is implemented by inserting a method defined as an *advice* into *join points* specified in a program. The SC language system also can be used to implement this feature defining such insertion by adequate transformation rules.

4.5 Pattern-matching

There exists many implementations of pattern-matching utilities on S-expressions[15]. But the patterns which correspond to ,@*symbol* and ,@*symbol*[*function-name*] are not so popular.

In most of implementations, the form ?*symbol* is used as a pattern which corresponds to our ,*symbol*. We adopted more intuitive backquote-macro-like notations in consideration of a symmetry between patterns and expressions.

4.6 Another S-Expression Based C

Symbolic C Expressions[16] (Scexp) is another S-expression based C language. It emphasizes usefulness of writing C code with Lisp macros. However, it is not intended to be a base for language extensions.

5. CONCLUSION AND FUTURE WORK

We proposed a scheme for extending the C language, where an S-expression based extended language is translated into a C language with an S-expression syntax. In this scheme, we can extend C at lower implementation cost because we can easily manipulate S-expressions using Lisp and transformation rules can be written intuitively. We also presented an practical example of a language extension which adds nested functions to C.

Future work includes a way how to apply (independently developed) two or more extensions. We will also implement high-level languages (e.g., with a garbage collected heap) based on LW-SC.

6. REFERENCES

[1] 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).

[2] Hiraishi, T., Li, X., Yasugi, M., Umatani, S. and Yuasa, T.: Language Extension by Rule-Based Transformation for S-Expression-Based C Languages, *IPSJ Transsactions on Programming*, Vol. 46, No. SIG1(PRO 24), pp. 40–56 (2005). (in Japanese).

[3] Tabata, Y., Yasugi, M., Komiya, T. and Yuasa, T.: Implementation of Multiple Threads by Using Nested Functions, *IPSJ Transsactions on Programming*, Vol. 43, No. SIG 3(PRO 14), pp. 26–40 (2002). (in Japanese).

[4] Stallman, R. M.: Using and Porting GNU Compiler Collection (1999).

[5] Breuel, T. M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).

[6] Frigo, M., Leiserson, C. E. and Randall, L.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices*, Vol. 33, No. 5, pp. 212–223 (1998).

[7] PHASE Editorial Committee: Omni: OpenMP compiler project. `http://phase.hpcc.jp/Omni/`.

[8] Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol. 13, No. 3, pp. 284–293 (1990).

[9] Joel, F. B.: SCHEME->C a Portable Scheme-to-C Compiler, *WRL Research Report* (1989).

[10] Kelsey, R.: Pre-Scheme: A Scheme Dialect for Systems Programming.

[11] Chiba, S. and Masuda, T.: Designing an Extensible Distributed Language with a Meta-Level Architecture, *In Proceedings of European Conference on Object Oriented Programming (ECOOP), LNCS 707*, pp. 482–501 (1993).

[12] Chiba, S.: A Metaobject Protocol for C++, *ACM Sigplan Notices*, Vol. 30, No. 10, pp. 285–299 (1995).

[13] Tatsubori, M., Chiba, S., Killijian, M.-O. and Itano, K.: OpenJava: A Class-based Macro System for Java, *Reflection and Software Engineering* (Cazzola, W., Stroud, R. J. and Tisato, F.(eds.)), LNCS 1826, Springer-Verlag, pp. 119–135 (2000).

[14] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proceedings European Conference on Object-Oriented Programming* (Akşit, M. and Matsuoka, S.(eds.)), Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp. 220–242 (1997).

[15] Queinnec, C.: Compilation of Non-linear, Second Order Patterns on S-expressions, *PLILP'90, LNCS 456*, pp. 340–357 (1990).

[16] Mastenbrook, B.: scexp — Symbolic C Expressions (2004). `http://www.unmutual.info/software/scexp/`.