Experience with SC: Transformation-based Implementation of Various Extensions to C

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

We have proposed the SC language system which facilitates language extensions by translation into C. In this paper, we present our experience with the SC language system and discuss its design, implementation, applications and improvements. In particular, we present the improvement to the design of transformation rules for implementing translations, which includes the feature to extend an existing transformation phase (rule-set). This enables us to implement many transformation rule-sets only by describing the difference, and helps us to use commonly-used rule-sets as part of the entire transformation. We also show several actual examples of extensions to C: garbage collection, multithreading and load balancing.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.3.4 [**Programming Languages**]: Software—*Translator writing systems and compiler generators*

General Terms

Languages, Design

1. INTRODUCTION

The C language is often indispensable for developing practical software systems. Sometimes extended C languages are suitable for elegant and efficient development. A language extension can be implemented by modifying a C compiler, but in many cases it can also be done by translating extended C programs into C.

We have proposed the SC language system which facilitates transformation-based language extensions[9]. SC languages are extended C languages with an S-expression-based syntax. A distinguished SC language, called SC-0, is the plain C language with a Lisp-like syntax. SC-0 is implemented by a translator to the C language. With the SC language system, a language extension is implemented as a translator to SC-0 (and then to C) (See Figure 1). Such a translator can be easily implemented for the following reasons:

- a program represented by S-expressions can be used as an AST (abstract syntax tree) without any preprocessing,
- we can use Lisp facilities for manipulating S-expressions, and
- we can handle environments (translation contexts) easily by using dynamic variables in Common Lisp.

So far, we have developed various language extensions using this system. In this process, we have also improved the system itself to solve several problems that we encountered. For example, we introduced a pattern matching facility over S-expressions for simplifying the notation of translating function definitions[9]. In addition, we implemented a translator from C to SC-0, which enables SC programs to use function declarations and macros in C header files[10].

We also encountered a significant problem that we had to modify the existing implementations when we used multiple transformation phases at the same time, resulting poor maintainability and reusability. To solve this problem, we devised a facility for extending code translators. This facility reduces duplicated description, enables differential programming, and manages existing implementations efficiently.

This paper also presents some high-level languages developed by using the SC language system. They are realized by using LW-SC as an intermediate language, which is an extended SC language with nested functions and we presented before[9, 11].

2. SC LANGUAGE SYSTEM

2.1 Overview

The SC language system is implemented in Common Lisp and consists of the following modules:

- the SC preprocessor, which includes specified SC files and handles macro definitions and expansions,
- the SC translator, which applies transformation rules and generates code into another SC language, and

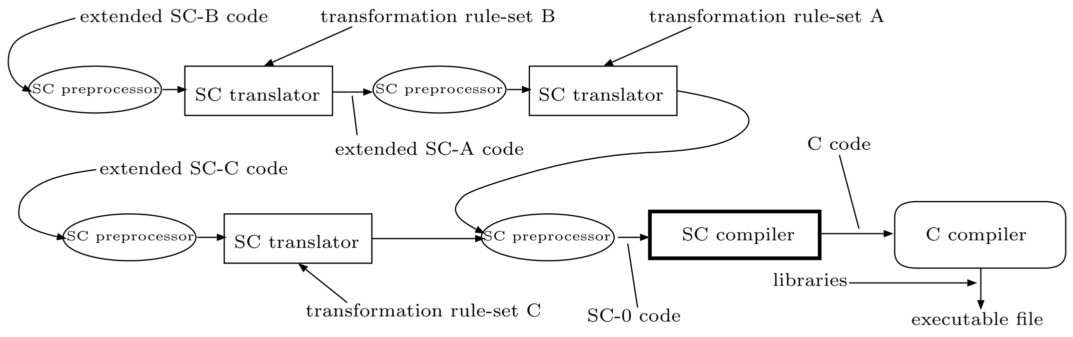

Figure 1: Code translation phases in the SC language system.

```
(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: A C program equivalent to Figure 2.

- the SC compiler, which compiles SC-0 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. At each translation phase, the SC preprocessor and then a transformation *rule-set* is applied. Extension implementers can develop a new translation phase simply by writing a new transformation rule-set.

Figure 2 shows an example program in SC-0, the final target of the translators. This program is equivalent to the C program in Figure 3. See [8] for the full syntax of SC-0.

2.2 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*$_n$`)`
  is 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, and then the result is evaluated by Common Lisp. If the return value `eql`s `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*`)`
  specifies a C header file *file-name*. 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/`#define`d in C header files[1].

2.3 Transformation rules

From the earlier version[9], we made some changes in the design of transformation rules. Now we define a translation phase as a rule-set, and introduced the concept of rule-set extension explicitly. In addition, we introduced more Lisp-like and less declarative notations.

[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[10].

2.3.1 Specification

Defining rule-sets. Each translation phase in Figure 1 is governed by a transformation rule-set.

A code transformation phase in Figure 1 is defined as a *rule-set.* The syntax for defining a rule-set is as follows:

```
(define-ruleset rule-set-name (parent-rule-set-name ...)
  (parameter-name default-value)
  ...
  ).
```

The *rule-set-name* specifies a name of the being defined rule-set. The rule-set extends zero or more rule-sets specified by *parent-rule-set-name.* The extended rule-set inherits the all rules and the all parameters from the parent rule-sets. Parameters, specified with *parameter-name* and *default-value*, can be referred to by rules which belong to the rule-set with the `ruleset-param` function.

In particular, parameters with the following names have special meanings.

- `entry`: The value must be a symbol and it specifies a rule which is called at the beginning when the rule-set is applied.
- `default-input-handler`: The value must be a function with one argument. This function is called if no pattern matches when a rule is applied.

Defining rules. Then we need to define *rule*s which belong to the rule-set. [2]

```
(defrule rule-name rule-set-name
  ((#?pattern_11 ...#?pattern_1m_1)
   form-list_1)
  ...
  ((#?pattern_n1 ...#?pattern_nm_n)
   form-list_n)
  [(otherwise form-list_otherwise)]
  )
```

When a rule is applied, the parameter is tested whether it is matched by any of pattern in the following order: *pattern*$_{11}$, ... *pattern*$_{1m_1}$, ... *pattern*$_{n1}$, ... *pattern*$_{nm_n}$. The form list *form-list*$_i$ is evaluated (by a Lisp evaluator) when the argument is matched by *pattern*$_{ik}$. If no pattern matches the parameter, *form-list*$_{\text{otherwise}}$ is evaluated if exists, otherwise the function `default-input-handler` of the rule-set *rule-set-name* is called (the parameter is passed to the function).

We can define a rule using keyword `extendrule` instead of `defrule`. `Extendrule` has the same syntax and semantics as `defrule` except that the rule *rule-name* of a parent rule-set is applied when no pattern matches and no `otherwise` clause is given. In the case where a rule-set has multiple parent rule-sets, the applying order is determined by the "class precedence list" of CLOS.

[2]Parentheses enclosing (a) *pattern*(s) can be omitted where $m_i = 1$.

Patterns. We specify *pattern*s using notations similar to backquote macros. More precisely, *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 element.

(3) `,@`*symbol*
matches zero or more any elements.

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

(5) `,@`*symbol*`[`*function*`]`
matches zero or more elements if the evaluation result of `(every #'`*function list*`)` is non-`nil`, where *list* is a list of the elements.

The function *function* can be what is defined as a transformation rule or an ordinary Common Lisp function (a built-in function or what is defined separately from transformation rules). A `lambda` special form can be written directly as a *function.*

In evaluating a form list in a `defrule` (`extendrule`) body, variable `x` is bound to the whole S-expression of the parameter and, in all the cases except (1), *symbol* is bound to the matched part of the S-expression. In addition, the `get-retval` function can be used to get an actual return value of *function* in (4) and (5) by passing the corresponding *symbol.*

Applying rule-sets. A defined rule-set (transformation phase) can be applied using the function `apply-ruleset`:

```
(apply-ruleset input :rule-set-name).
```

The `apply-ruleset` receives additional keyword parameters corresponding to *parameter-name*s, which change values of rule-set parameters. Here shows an example:

```
(apply-ruleset ~(* a b) :sc0-to-sc0
               :entry 'expression).
```

This function applies the `expression` rule of the specified rule-set `sc0-to-sc0` to *input.*[3] In processing `apply-ruleset`, dynamic variable `*current-ruleset*` which stands for the *current rule-set* is dynamically bound to the specified rule-set.

[3]The *input* can be a string or a pathname which specifies a SC source file.

```
(define-ruleset sc0-to-sc0 ()
  (entry 'sc-program)
  (default-input-handler #'no-match-error))
(defrule sc-program sc0-to-sc0
  (#?(,@decl-list)
   (mapcar #'declaration decl-list)) )
(defrule declaration sc0-to-sc0
  (#?(,scs[storage-class-specifier] ;function definitions
       (,@id-list[identifier]) (fn ,@tlist) ,@body)
   ~(,scs (,@id-list) ,(third x)
       ,@(mapcar #'block-item body)))
  ...)
(defrule block-item sc0-to-sc0
  ((#?,bi[declaration]
    #?,bi[statement])
   (get-retval 'bi))
  )
(defrule statement sc0-to-sc0
  (#?(do-while ,exp ,@body)
   ~(do-while ,(expression exp)
      ,@(mapcar #'block-item body)))
  ...
  (otherwise (expression x)) ;expression-statements
  )
...

(define-ruleset sc1-to-sc0 (sc0-to-sc0))
(extendrule statement sc1-to-sc0
  ((#?(let (,@decl-list) ,@body) )
   ~(begin ,@(mapcar #'declaration decl-list)
           ,@(mapcar #'block-item body)) )
  ((#?(while ,exp ,@body) )
   (let ((cdt (expression exp)))
     ~(if ,cdt
          (do-while ,cdt ,@(mapcar #'block-item body))) ))
  ((#?(for (,@list ,exp2 ,exp3) ,@body) )
   (let ((e1-list (mapcar #'block-item list))
         (e2 (expression exp2))
         (e3 (expression exp3))
         (new-body (mapcar #'block-item body)))
     (list ~(begin
             ,@e1-list
             (if ,e2
                 (do-while (exps ,e3 ,e2)
                   ,@new-body))))
     ) )
  ((#?(loop ,@body) )
   ~(do-while 1 ,@(mapcar #'block-item body)))
  )
```

Figure 4: Transformation rule-sets.

Applying rules. A rule defined with `defrule` or `extendrule` can be called as a function which takes one required argument and optional ones: the required one is an input for the rule, and the optional ones can specify a rule-set name and values of its parameters to supersede defaults. If a rule-set name is given explicitly, the rule of the specified rule-set is applied to the input and the *current rule-set* is bound to the specified rule-set. Otherwise, the rule of the *current rule-set* is applied.

As described above, we can refer to parameters of a rule-set with the `ruleset-param` function. This function takes the symbol of a parameter name and returns the value of the parameter of the current rule-set.

2.3.2 Example

Figure 4 shows an example of definitions of transformation rule-sets. A character '~' in this figure is a macro character which works as if it were a backquote, except that symbols in the following expression are interned to a distinguished package for SC code instead of the current package (`*package*`) [4]. Two rule-sets are defined in this figure. The `sc0-to-sc0` rule-set defines a identical transformer from SC-0 to SC-0. The `sc1-to-sc0` rule-set defines a transformation from the SC-1 language to SC-0. SC-1 features several constructs for iteration and bindings added to SC-0.

Suppose that the form

```
(statement ~(do-while x (while y (++ z)))
           :sc1-to-sc0)
```

is evaluated. This evaluation is progressed as follows:

1. The `statement` rule of `sc1-to-sc0` is applied to the given input; no pattern matches.
2. The `statement` rule of `sc0-to-sc0` is applied; the pattern `#?(do-while ...)` matches.
3. Because the `expression` rule of `sc1-to-sc0` is not defined, the `expression` rule of `sc0-to-sc0` is applied to `x`; `x` is returned. (the `expression` rule is snipped in the figure.)
4. Because the `block-item` rule of `sc1-to-sc0` is not defined, the `block-item` rule of `sc0-to-sc0` is applied to `(while y (++ z))`, and then the `statement` rule of `sc1-to-sc0` is applied; `(if y (do-while y (++ z)))` is returned.
5. `(do-while x (if y (do-while y (++ z))))` is returned.

Since the current rule-set varies according to context, the rule of the current rule-set also varies. For example, when `(statement ...:sc0-to-sc0)` is evaluated, the `statement` rule of `sc0-to-sc0` is applied to `(while ...)` and `no-match-error` occurs.

2.3.3 Implementation

This rule-set extension facility is implemented using CLOS. Figure 5 shows (simplified) implementation code for `define-ruleset`, `defrule`, `extendrule` and `apply-ruleset`. Roughly speaking, `define-ruleset` and `defrule` (or `extendrule`) correspond to `defclass` and `defmethod` respectively so that we can dispatch an appropriate rule for the being applied rule-set.

In addition, the dynamic variable `*current-ruleset*` is used to remember the being applied rule-set and each function to which a rule name is bound wraps the actual method call with the value of this dynamic variable. As a result, we need not to write the trivial rule-set argument in the definitions.

3. DISCUSSION

In the earlier version of the SC system, the definition of transformation rules which correspond to Figure 4 can be written as in Figure 6. All the rules were at the same

[4]Though the `keyword` package is generally used for treating shared symbols across multiple packages, we employed this notation to avoid writing a preamble '`:`' for every symbol.

```
(defmacro define-ruleset (name parents &body parameters)
  `(defclass ,(ruleset-class-symbol name)
         ,(or (mapcar #'ruleset-class-symbol parents)
              (list *base-ruleset-class-name*))
       ,(loop for (p v) in parameters
           collect `(,p :initform ,v
                        :initarg ,p))))

(defun apply-rule (input ruleset &rest initargs)
  (let ((*current-ruleset*
         (apply #'make-instance ruleset initargs)))
    (funcall (rule-function
              (slot-value *current-ruleset* 'entry))
             (if (or (stringp input)
                     (pathnamep input))
                 (sc-file:read-sc-file input)
                 input))))

(defun rulemethod-args (ruleset)
  `(x (,*ruleset-arg* ,(ruleset-class-symbol ruleset))))
(defmacro defrule (name ruleset &body pats-act-list)
  `(progn
     (defmethod ,(rule-method-symbol name)
                ,(rulemethod-args ruleset)
       (block ,name
         (case-match x
          ,@pats-act-list
          (otherwise ;(call-next-method) in extendrule.
           (call-otherwise-default x ',ruleset)))))
     (defun ,(rule-function-symbol name)
         (x &optional (ruleset *current-ruleset* r)
            &rest initargs)
       (if r
           (let ((*current-ruleset*
                  (apply #'make-instance ruleset initargs)
             (,(rule-method-symbol name) x *current-ruleset*))
           (,(rule-method-symbol name) x *current-ruleset*)))))
```

Figure 5: Implementation code for defining rule-sets.

level and the concept of "rule-set" did not exist explicitly. (Though we used this term, which simply indicated a group of rules which was related to one transformation phase.) Therefore we had to write rules for the whole syntax, even if an extended language has only several new constructs.

3.1 Extensibility of rule-sets

In the current version, existing rule-sets can be extended to define a new rule-set. In Figure 4, the `sc1-to-sc0` rule-set is defined by extending `sc0-to-sc0`, which is a rule-set for identical transformation. By using `sc0-to-sc0` as a common template, many transformation rule-sets can be defined only by describing the difference.

Extending rule-sets is also helpful when we use a commonly-used rule-set as part of the entire transformation. For instance, when we implement high-level services, we divide the entire transformation into several phases. Indeed, when we implemented MT-SC (SC-0 with multithreading), we used the `type-info` rule-set to add type information to all expressions. In this case we cannot use the original `type-info` rule-set as it is, but we can easily extend it for MT-SC (See Figure 7). In a similar manner, we can easily reuse the `type-info` rule-set for implementing other extended languages by writing a small amount of additional code.

In addition, extending rule-sets is also helpful when we want to use multiple extensions at the same time. Suppose we have already implemented extensions A and B, and we want an extended language which features both of them. As Figure 8 shows, such an implementation could be done by connecting two rule-sets serially one after another. If we choose the left path, we must modify the rule-set for B to support new features added in A. This modification can be done by the rule-set extension facility described above. We can write the difference separately instead of rewriting the rule-set B. This scheme works well if A is a relatively simple extension (e.g. new constructs for iteration). However if both A and B are non-trivial extensions, extending a rule-set is not easy because the semantics for A+B is not straightforward.

```
(sc0-program (,@decl-list))
-> (mapcar #'declaration decl-list)) )
(sc0-declaration (,scs[storage-class-specifier]
                  (,@id-list[identifier]) (fn ,@tlist) ,@body))
-> ~(,scs (,@id-list) ,(third x)
     ,@(mapcar #'block-item body)))
...
(sc0-block-item ,bi[declaration])
(sc0-block-item ,bi[statement])
-> (get-retval 'bi)

(sc0-statement (do-while ,exp ,@body))
-> ~(do-while ,(expression exp)
      ,@(mapcar #'block-item body)))
...
(sc0-statement ,otherwise)
-> (expression x)
...

(sc1-program (,@decl-list)) -> ...
(sc1-declaration (,scs[storage-class-specifier] ...)) -> ...
(sc1-block-item ,bi[declaration])
(sc1-block-item ,bi[statement])   -> ...

(sc1-statement (do-while ,exp ,@body)) -> ...
...
(sc1-statement (let (,@decl-list) ,@body))
-> ~(begin ,@(mapcar #'declaration decl-list)
           ,@(mapcar #'block-item body))
(sc1-statement (while ,exp ,@body))
-> (let ((cdt (expression exp)))
     ~(if ,cdt
          (do-while ,cdt ,@(mapcar #'block-item body))) )
(sc1-statement (for (,@list ,exp2 ,exp3) ,@body))
-> (let ((e1-list (mapcar #'block-item list))
         (e2 (expression exp2))
         (e3 (expression exp3))
         (new-body (mapcar #'block-item body)))
     (list ~(begin
             ,@e1-list
             (if ,e2
                 (do-while (exps ,e3 ,e2)
                   ,@new-body)))))
(sc1-statement (loop ,@body))
-> ~(do-while 1 ,@(function-body body))
```

Figure 6: The earlier version of transformation rules.

Figure 9 shows code for extending the `sc1-to-sc0` rule-set to fit MT-SC. Because such differential code can be managed independently of base rule-sets, we can manage rule-sets more clearly.

3.2 Ease of use

In the old version, we used to introduce more declarative notation for transformation rules in consideration of other rule-based languages such as Prolog. Now transformation rules are described with ML-like notations for pattern matching,

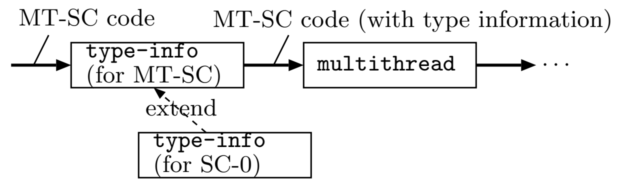

Figure 7: A rule-set as part of the entire transformation.

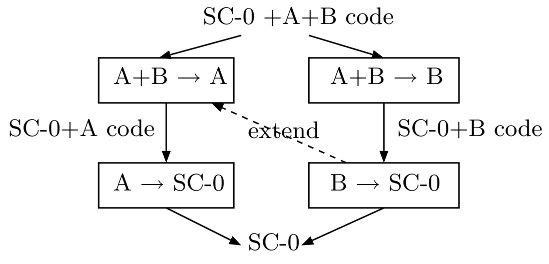

Figure 8: Applying multiple rule-sets.

mainly because of implementation convenience. Some people may prefer the old syntax, but we think the new one is more intuitive because rules look like BNF notations.

4. IMPLEMENTING EXTENDED LANGUAGES

The SC language system allows language extensions from simple ones to non-trivial ones. Examples of relatively simple extensions include some new constructs for iterations and Java-like labeled-breaks/continues[8]. On the other hand, non-trivial extensions include exception handlers[16] and nested functions, and their implementations typically need additional declarations of temporary variables; thus type information is also required.

In earlier work, we implemented LW-SC, an extended SC language with nested functions[9, 11]. Without returning from a function, a nested function can manipulate its caller's local variables (or local variables of its indirect callers) by indirectly calling a nested function of its (indirect) caller. Thus, many high-level services with "stack walk" (such as check-pointing and copying GC) 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 cost of creating and maintaining nested functions.

We have also implemented nested functions with lightweight closures by modifying the GCC compiler[19], with which we can get better performance.

4.1 HSC — Copying GC

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

```
(define-ruleset multithread-sc1 (sc1-to-sc0))

(extendrule statement multithread-sc1
  (#?(thread-create ,dec-list ,@body)
   ~(thread-create
      ,(mapcar #'declaration dec-list)
      ,@(mapcar #'block-item body)) )
  (#?(thread-suspend ,id[identifier] ,@body)
   ~(thread-suspend ,id ,@(mapcar #'block-item body)) )
  (#?(thread-resume ,exp)
   ~(thread-resume ,(expression exp)))
  )
```

Figure 9: Extending the `sc1-to-sc0` rule-set for MT-SC.

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 distinguish *roots* from other elements in the stack. For this reason, conservative collectors [2] are often 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. But as described above, we can implement a garbage collector with low maintenance costs using lightweight nested functions.

By embedding garbage collection, we actually implemented *HSC* (High-level SC), which is a memory safe SC-0 language with objects allocated in a garbage collected heap.

4.1.1 Specification

To guarantee memory safety, we modified the specification of SC-0. In particular, HSC uses "references" instead of pointers. Therefore,

- getting addresses of variables by `ptr` (`&` in C),
- arithmetic operations for references, and
- pointer casts

are not permitted in HSC. The other modifications are as follows:

- The syntax `(new` *expression*`)` is added to *expression*, which evaluates a given expression, allocate an object initialized to the result, and then returns a reference to the object.
- An array type is not equivalent to any pointer types even if it appears as an argument type.
- An array reference
  `(aref` *expression*$_1$ *expression*$_2$`)`
  is permitted only if the value of *expression*$_1$ has an array type. It is no longer equivalent to
  `(mref (+` *expression*$_1$ *expression*$_2$`))`.
- No `union` types exist. (From a practical viewpoint, disjoint unions should be implemented instead.)

```
(def (struct sPair)
  (def car (ptr Object))
  (def cdr (ptr Object)))
(deftype Pair (struct sPair))

(def (make-pair e1 e2)
     (fn (ptr Pair) (ptr Object) (ptr Object))
  (def pair (ptr Pair) NULL)
  (counter-on)
  (= pair (new (init Pair (struct e1 e2))))
  (counter-off)
  (return pair))
```

Figure 10: An HSC program.

```
(deftype sht (ptr (lightweight void void)))

(def Pair-desc (struct descriptor)
  (struct size map-array ...))

(def (make-pair scan0 e1 e2)
     (fn (ptr Pair) sht (ptr Object) (ptr Object))
  (def pair (ptr Pair) NULL)
  (def (scan1) (lightweight void void)
    (= e1 (evacuate e1)) (= e2 (evacuate e2))
    (= pair (evacuate pair))
    (scan0))
  (counter-on scan1)
  (= pair (getmem scan1 (ptr Pair-desc)))
  (= (mref pair) (init Pair (struct e1 e2)))
  (counter-off scan1)
  (return pair))
```

Figure 11: Scanning stack implemented by nested functions in LW-SC.

Figure 10 shows an example of an HSC program. In this figure, `(init Pair (struct e1 e2))` is an SC-0 expression which is equivalent to `(Pair){e1,e2}` in C99[12].

4.1.2 Implementation

Figure 11 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 evacuation (copying) of objects by using roots (`e1`, `e2` and `pair`) 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.

4.2 MT-SC — Multithreading

We implemented an extended SC-0 language *MT-SC* involving features for multithreading.

We used the implementation techniques which we proposed before in [15].

4.2.1 Specification

By "threads", we do not mean OS threads; we mean language-level threads. Each thread of MT-SC is "active" or "suspended". A thread is created by `thread-create` statement in "active" state. The thread can suspend itself to become "suspended", and at the same time a continuation of the thread can be saved. Another thread can resume the suspended thread by specifying the continuation. A thread is eliminated when it finishes the given computation.

```
(def (pfib n) (fn int int)
  (def x int) (def y int)
  (def nn int 0) (def c cont 0)
  (if (<= n 1)
      (return 1)
      (begin
       (thread-create
         (= x (pfib (- n 1)))
         (if (== (++ nn) 0)
             (thread-resume c))) ;  Resume the waiting thread.
       (= y (pfib (- n 2)))
       (if (< (-- nn) 0) ;  Wait for synchronization.
         (thread-suspend c0 (= c c0)))
       (return (+ x y)))))
```

Figure 12: An MT-SC program.

MT-SC has following primitives:

- `(thread-create` *body*`)` creates a new thread which executes *body*,
- `(thread-suspend` *identifier body*`)` binds the variable *identifier* to the current continuation, executes *body* to save the continuation, and then makes the current thread suspended, and
- `(thread-resume` *expression*`)` resumes the suspended thread. The value of *expression* should be a continuation saved by `thread-suspend`.

Figure 12 shows an example of an MT-SC program.

4.2.2 Implementation

These features can be implemented using nested functions in LW-SC as shown in Figure 13. Every function has its own nested function to continue its equivalent computation and saves the pointer of the nested function to be called later to early execute the thread's unprocessed computation (continuation). Such a nested function is also generated for each `thread-create`.

A translated program also includes a scheduler function `scheduling` and a thread stack. The thread stack holds a state and a continuation (a pointer of nested function) corresponding to each thread. When the scheduler is called, it takes one of active threads and resumes it by calling its nested function.

Our implementation does not need per-thread execution stacks, and nor heap memory for storing thread frames.

4.3 T-Cell — Automatic loads balancing

Today parallel computing becomes popular. However, various programming models for shared memory environments (e.g., multi-core processors, multi-processors) or distributed memory environments (e.g., clusters, grid computing systems) confuse many programmers.

To address this problem, we are developing a load-balancing framework based on lazy partitioning where workers automatically exchange tasks with each other via base servers, and the *T-Cell* language for the framework. Lazy partitioning means that tasks are partitioned only when a request has arisen.

```
(decl (struct _thstelm))
(deftype cont (ptr (lightweight (ptr void)
                      (ptr (struct _thstelm)) reason)))
(def (struct _thstelm)
  (def c cont)
  (def stat (enum t-stat)))
(deftype thst_ptr (ptr (struct _thstelm)))
(def thst (array (struct _thstelm) 4192)) ;  a thread stack
(def thst_top thst_ptr thst) ;  top of the thread stack

(def (pfib c_p n) (fn int cont int)
 (def ln int 0)
 (def x int) (def y int)
 (def nn int 0) (def c thst_ptr 0)  (def c0 thst_ptr)
 (def tmp2 int) (def tmp1 int)

 (def (pfib_c cp rsn)
      (lightweight (ptr void) thst_ptr reason)
   (switch rsn
     (case rsn_cont)
       (switch ln
         (case 1) (goto L1)
         (case 2) (goto L2)
         (case 3) (goto L3))
       (return)
     (case rsn_retval)
       (switch ln
         (case 2)
           (return (cast (ptr void) (ptr tmp2))))
       (return))
   (return)
   ...  Almost the same contents as the owner function ...
   )

 (if (<= n 2)
     (return 1)
     (begin
       ;;  push a current continuation to the thread stack
       (begin
         (= ln 1)
         (= (fref thst_top -> c) pfib_c)
         (= (fref thst_top -> stat) thr_new_runnable)
         (inc thst_top))
       ;;  the body of  thread-create
       (begin
         (def ln int 0)
         (def (nthr_c cp rsn)
              (lightweight (ptr void) thst_ptr reason)
           ...)
         (= ln 1)
         (= x (pfib nthr_c (- n 1)))
         (inc nn)
         (if (== nn 0) (thr_resume c)))
       ;;  pop the thread stack
       (if (!= (fref (- thst_top 1) -> stat)
               thr_new_runnable)
           (scheduling)      ;  call a scheduler
           (dec thst_top))
       ;; (label L1)  (in the nested function)
       (= ln 2)
       (= y (pfib pfib_c (- n 2)))
       ;; (label L2)  (in the nested function)
       (= nn (- nn 1))
       (if (< nn 0)
           (begin
             ;;  suspend a current thread
             (= c0 (inc thst_top))
             (= (fref c0 -> c) pfib_c)
             (= (fref c0 -> stat) thr_new_suspended)
             (= c c0)
             (= ln 3)
             (scheduling)))  ;  call a scheduler
       ;; (label L3)  (in the nested function)
       (return (+ x y)))))
```

Figure 13: Multithreading implemented by LW-SC.

Figure 14 shows a multi-stage overview of the load-balancing framework. Compiled T-Cell programs are executed on one or more computation nodes. Each computation node has one or more workers in the shared memory environment.

Workers can communicate with each other by message passing. For automatic loads balancing, idle workers request tasks of busy workers. To relay a task request message, the base server which received the task request guesses a busy node and passes the request to it, maybe via another base server.

To spawn a larger task, the busy worker which received a task request temporarily backtracks its computation as far as possible before spawning a new task. For example, suppose that a worker computes fib(n), the n-th Fibonacci number, with the program in Figure 15. The worker begins with computing fib($n-1$). If the worker receives a task request in computing fib($n-1$), it spawns a task for computing fib($n-2$). Such mechanism can be implemented by using nested functions[20, 21].

Unlike LTC[13] and Cilk[5], our approach does not create multiple logical threads as potential tasks nor manage any queue of them. We employ a sequential program with task spawn handlers, which can be regarded as restartable exception handlers.

4.3.1 Specification

Figure 15 shows an example of a T-Cell program. Programmers can write a worker program with new constructs in T-Cell based on an existing sequential program. T-Cell has constructs for defining a task and for specifying potential task creation and result use.

The structure of a task object can be defined in the same way as `struct`, except that we may specify an `:in` or `:out` attribute for each field.

The computation of a task is defined by
`(def (task-body` *task-name*`)` *body*`)`.
A worker executes *body* when it receives the task *task-name*. In a `task-body` body, we can refer to a task object by the keyword `this`, which includes an input of the task in `:in` fields, and we should set the result of the computation into `:out` fields.

A statement
`(do-two` $statement_1$ $statement_2$
  `(`*task-name* `(:put` $body_{put}$`) (:get` $body_{get}$`)))`
indicates that the computation of $statement_1$ and $statement_2$ is divisible. This execution progresses as follows:

1. The statement $statement_1$ is executed. If the task request handler for this `do-two` statement is invoked, the current task is divided by spawning a new task, setting the fields of the task object by $body_{put}$, and then sending it to the task requester. Here, the computation to be done by $statement_2$ is packed as the task object.
2. If the task request handler for this `do-two` statement is not invoked until the execution of $statement_1$ is finished, the statement $statement_2$ is then executed. Oth-

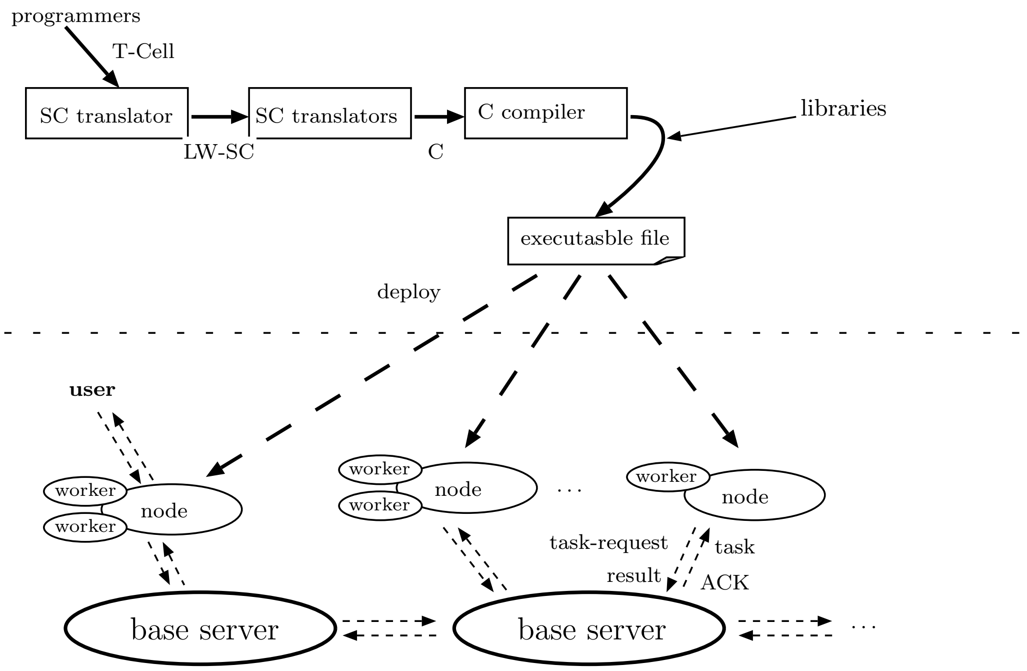

Figure 14: A load-balancing framework based on lazy partitioning.

erwise, the execution of *statement*$_2$ is skipped and the worker waits for the result of the spawned task and then merges the result by executing *body*$_{get}$. In waiting for the result of the spawned task, the worker may execute another task.

The identifier *task-name* specifies the type of task to be created. As in a `task-body` body, the keyword `this` can be used to refer to the task object in *body*$_{put}$ and *body*$_{get}$. We should set an input for a task in *body*$_{put}$ by assigning the value to `:in` fields, and can get a result in *body*$_{get}$ by referring to `:out` fields. This series of operations should be equivalent to the execution of *statement*$_2$.

The availability of the task request handler for a `do-two` statement has a dynamic extent and multiple handlers for `do-two` statements can be nested. The oldest handler should be invoked when a worker received a task request in order to make spawned tasks as large as possible.

For dividing iterative computation, T-Cell has the `do-many` construct. The syntax is as follows:

```
(do-many for identifier
         from expression_from to expression_to
  body
  (task-name
    (:put from identifier_from to identifier_to body_put)
    (:get body_get)))
```

This iterates *body* over integers from *expression*$_{from}$ to *expression*$_{to}$. When the task request handler is invoked, a part of iterations is spawned as a new task. The actual assigned range can be referred to in *body*$_{put}$ via *identifier*$_{from}$ and *identifier*$_{to}$.

In addition, T-Cell has "undo-redo clauses", syntactically denoted by

```
(dynamic-wind (:before body_before)
              (:body body)
              (:after body_after)).
```

This executes *body*$_{before}$, *body* and *body*$_{after}$ in this order. The *body*$_{after}$ is also executed before an attempt to divide older `do-two/do-many` computation, *body*$_{before}$ is also executed after the attempt[5]. By using this construct, we can avoid undesirable copying of temporarily-modified data (e.g., for backtrack search problems) and promote reuse/sharing of the working space.

4.3.2 Implementation

Such lazy partitioning can be implemented using nested functions as shown in Figure 16. Each `do-two` statement is translated into a piece of code which includes a definition of a nested function, and each function is translated to have an additional argument `-bk0`. This additional argument holds a nested function pointer corresponding to the newest handler for `do-two/do-many` statements, which is called when a task request is detected by polling. The nested function firstly tries larger task spawning by calling a nested function which corresponds to an one more older handler. Only if a task request still remains, a new task is created and sent to the requester. After sending a task, a worker returns from the nested function and resumes its own computation.

The nested function composed of *body*$_{after}$ and *body*$_{before}$ can be used to implement a `dynamic-wind` statement. The translator also generates functions for sending/receiving and se-

[5]This construct can be considered as a restartable exception handler version of the "try-finally" construct.

```
(def (task tfib)
  (def n int :in)
  (def r int :out))

(def (task-body tfib)
  (= (fref this r)
     (fib (fref this n))))

(def (fib n) (wfn int int)
  (if (<= n 2)
      (return 1)
    (begin
      (def s1 int) (def s2 int)
      (do-two
          (= s1 (fib (- n 1)))
          (= s2 (fib (- n 2)))
        (tfib
         (:put (= (fref this n) (- n 2)))
         (:get (= s2 (fref this r)))))
      (return (+ s1 s2)))))
```

Figure 15: A T-Cell program for Fibonacci.

```
(def (fib -bk0 -thr n)
     (fn int (ptr (lightweight int)) (ptr thread) int)
  (if (<= n 2) (return 1))
  (begin
    (def s1 int) (def s2 int)
    (def this -task-tfib)
    (def spawned int 0)
    (begin
      (def (-bk) (lightweight int) ;nested function
        (if spawned (return 0))
        (-bk0) ; the nested function of the caller
        (if (try-to-get-treq -thr)
            (= (fref this n) (- n 2))  ; body_put
            (= spawned 1)
            (make-and-send-task -thr 0 (ptr this))
            (return 1))
        (return 0))
      (if (fref -thr -> req)            ; polling
          (-bk))                        ; start backtracking
      (= s1 (fib -bk -thr (- n 1))))    ; statement_1
    (if spawned
        (begin
          (wait-rslt -thr)
          (= s2 (fref this r)))         ; body_get
        (= s2 (fib -bk0 -thr (- n 2)))) ; statement_2
    (return (+ s1 s2))))
```

Figure 16: Lazy task partitioning implemented by nested functions.

rializing/deserializing task inputs/outputs for each defined task.

Task requests can be sent from the same or another computation node. If a request is from the same node, data of a task and its result can be passed quickly via shared memory, otherwise they are transmitted as serialized messages via base servers. Because the choice is determined automatically, programmers need not to take care of memory environments.

5. RELATED WORK

See also our previous paper [9] for other language extensions to C, lower-level Scheme, reflection, aspect-oriented programming, other implementations of pattern-matching over S-expressions and another S-expression based C.

5.1 Other rule-based transformations

5.1.1 Expert systems

Traditionally expert systems[1] are well known as rule based solution systems. They use conflict resolution strategies which deal with more complicated cases.

Although our strategy only uses information about written orders of patterns and the current rule-set, we may employ *annotations* for transformation rule-sets to generate more sophisticated code (e.g., optimized code).

5.1.2 Rewriting rules

There are program transformation systems that use rewrite rules[17, 18, 3]. In most such systems, rules are defined more declaratively and environments for object languages are included in both patterns and outputs, while our translators treat such environments basically by using side-effects or dynamic bindings in Common Lisp.

Though we prefer our approach in perspective of intuitive implementations of transformations, it is also possible to use as a framework for implementing rewrite rules by defining side-effect-free rule-sets.

5.2 Programs as Lisp macro forms

Our system treats SC programs as data. On the other hand, there are some S-expression based markup languages where programs themselves are evaluated by a Lisp evaluator as macro forms and object code is generated as a result[14, 4].

Though such an implementation for SC translators is not impossible, it would not fit to the SC syntax well because the translator needs to change valid rules depending on contexts and most of Lisp macro facilities (including `define-syntax` of Scheme etc.) support only lists as patterns.

6. CONCLUDING REMARKS

Using SC language system, we have developed many extended C languages, and in this process we acquired some knowledge. For example, we realized that there exists transformations that can be commonly-used (e.g., adding type information or declarations of variables) and that they can be reused more efficiently with extensible rule-sets. Now we actually exploit this system for prototyping newly-invented languages and testing new implementation techniques, which can be done very easily. This system can also be used for implementing languages other than extended SC languages (e.g., Scheme and Java), which we plan to do as future work.

7. 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 MEXT Grant-in-Aid for Exploratory Research (17650008).

8. REFERENCES

[1] D. G. Bobrow, S. Mittal, and M. J. Stefik. Expert systems: perils and promise. *Commun. ACM*, 29(9):880–894, 1986.

[2] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, 18(9):807–820, 1988.

[3] T. Cleenewerck and T. D'Hondt. Disentangling the implementation of local-to-global transformations in a

rewrite rule transformation system. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1398–1403. ACM Press, 2005.

[4] Franz Inc. HTML generation facility. `http://allegroserve.sourceforge.net/aserve-dist/doc/htmlgen.html`.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices (PLDI '98)*, 33(5):212–223, 1998.

[6] D. R. Hanson and M. Raghavachari. A machine-independent debugger. *Software – Practice & Experience*, 26(11):1277–1299, 1996.

[7] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proc. of the 3rd International Symposium on Memory Management*, pages 150–156, 2002.

[8] T. Hiraishi, X. Li, M. Yasugi, S. Umatani, and T. Yuasa. Language extension by rule-based transformation for s-expression-based c languages. *IPSJ Transactions on Programming*, 46(SIG1(PRO 24)):40–56, 2005. (in Japanese).

[9] T. Hiraishi, M. Yasugi, and T. Yuasa. Implementing S-expression based extended languages in Lisp. In *Proceedings of the International Lisp Conference*, pages 179–188, Stanford, CA, 2005.

[10] T. Hiraishi, M. Yasugi, and T. Yuasa. Effective utilization of existing C header files in other languages with different syntaxes. *Computer Software*, 23(2):225–238, 2006. (in Japanese).

[11] T. Hiraishi, M. Yasugi, and T. Yuasa. A transformation-based implementation of lightweight nested functions. *IPSJ Digital Courier*, 2:262–279, 2006. (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50-67.).

[12] ISO/IEC. ISO/IEC 9899:1999(E) programming languages — C. 1999.

[13] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[14] K. Rosenberg. LML: The Lisp markup language. `http://lml.b9.com/`.

[15] Y. Tabata, M. Yasugi, T. Komiya, and T. Yuasa. Implementation of multiple threads by using nested functions. *IPSJ Transactions on Programming*, 43(SIG 3(PRO 14)):26–40, 2002. (in Japanese).

[16] S. Umatani, H. Shobayashi, M. Yasugi, and T. Yuasa. Efficient and portable implementation of Java-style exception handling in C. *IPSJ Digital Courier*, 2:238–247, 2006. (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 1-10.).

[17] E. Visser. Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357+, 2001.

[18] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[19] M. Yasugi, T. Hiraishi, and T. Yuasa. Lightweight lexical closures for legitimate execution stack access. In *Proceedings of 15th International Conference on Compiler Construction (CC2006)*, number 3923 in Lecture Notes in Computer Science, pages 170–184. Springer-Verlag, 2006.

[20] M. Yasugi, T. Komiya, and T. Yuasa. Dynamic load balancing by using nested functions and its high-level description. *IPSJ Transactions on Advanced Computing Systems*, 45(SIG 11(ACS 7)):368–377, 2004. (in Japanese).

[21] M. Yasugi, T. Komiya, and T. Yuasa. An efficient load-balancing framework based on lazy partitioning of sequential programs. In *Proceedings of the Workshop on New Approaches to Software Construction*, pages 65–84, 2004.