Journal of Information Processing Vol.27 177–189 (Feb. 2019)

[DOI: 10.2197/ipsjjip.27.177]

Regular Paper

Evaluating Portable Mechanisms for Legitimate Execution Stack Access with a Scheme Interpreter in an Extended SC Language

Masahiro Yasugi[1,a)] Reichi Ikeuchi[1,†1] Tasuku Hiraishi[2] Tsuneyasu Komiya[3]

Received: June 30, 2018, Accepted: September 26, 2018

Abstract: Scheme implementations should be properly tail-recursive and support garbage collection. To reduce the development costs, a Scheme interpreter called JAKLD, which is written in Java, was designed to use execution stacks simply. JAKLD with interchangeable garbage collectors was reimplemented in C. In addition, we have proposed an efficient C-based implementation written in an extended C language called XC-cube, which features language mechanisms for implementing high-level programming languages such as "L-closures" for legitimate execution stack access, with which a running program/process can legitimately access data deeply in execution stacks (C stacks). L-closures are lightweight lexical closures created from nested function definitions. In addition to enhanced C compilers, we have portable implementations of L-closures, which are translators from an extended S-expression based C language into the standard C language. Furthermore, we have another mechanism for legitimate execution stack access, called "closures". Closures are standard lexical closures created from nested function definitions. Closures can also be implemented using translators. In this study, JAKLD was reimplemented in an extended SC language (S-expression based C language) that features nested functions to evaluate (L-)closures and their implementations, including translators.

Keywords: interpreters, proper tail recursion, execution stacks, closures, transformation

1. Introduction

Implementations of the Scheme language [1], [12] are required to be properly tail-recursive and to support an unbounded number of active (pre-return) tail calls. For example, we can elegantly define a tail-recursive procedure (function) `my-gcd` that calculates the greatest common divisor based on the Euclidean algorithm as follows.

```
(define (my-gcd a b)
  (if (= b 0) a (my-gcd b (remainder a b))))
```

In addition, Scheme implementations are required to support garbage collection (called GC for short).

In a Scheme interpreter called "JAKLD" [19] that is written in Java, as its basic design, compilation is not required and execution stacks are simply used; this design makes it easy to maintain and extend the Scheme implementation. In JAKLD, built-in functions and special forms can be directly implemented. Although JAKLD was, at first, designed and developed as a Lisp driver to be embedded in Java applications, it can also be used as a stand-alone Scheme implementation. Later, JAKLD was reimplemented in C. The implementation (called "JAKLD/C" in this paper) consists of an interpreter and a memory manager. It features a customizable memory manager so that it can serve as a GC research platform.

Early in the development history, JAKLD and JAKLD/C were not properly tail-recursive. Later, JAKLD with trampolines was developed as a properly tail-recursive implementation [20]. JAKLD/C with trampolines can also be developed as a properly tail-recursive implementation.

Based on JAKLD/C, we have proposed a properly tail-recursive implementation [18] (called "JAKLD/XC" in this paper) written in an extended C language XC-cube [15], [16], which features mechanisms for implementing high-level languages, such as "L-closures" and "closures". *L-closures* and *closures* are mechanisms for legitimate execution stack access (called *LESA* for short), with which a running program/process can legitimately access data deeply in execution stacks (C stacks). The implementation scheme of JAKLD/XC is based on the key idea that is to avoid stack overflow by creating a space-efficient first-class continuation represented as a list containing only the "Frame" objects necessary for the rest of the computation and immediately invoking the continuation. In our previous work [18], this scheme has shown better performance than the trampoline based properly tail-recursive implementation of JALKD/C. By using LESA mechanisms for directly scanning GC roots, JAKLD/XC has shown better performance [18] than JAKLD/C with (push/pop

[1] Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[2] Academic Center for Computing and Media Studies, Kyoto University, Kyoto 606–8501, Japan
[3] Graduate School of Informatics and Engineering, The University of Electro-Communication, Chofu, Tokyo 182–8585, Japan
[†1] Presently with UDOM Co., Ltd.
[a)] yasugi@ai.kyutech.ac.jp

Table 1 The Scheme interpreters appearing in this paper. These interpreters simply use execution stacks of their implementation language.

Interpreter	Implementation language	Garbage collection	Proper tail recursion	First-class continuation
JAKLD [19]	Java	by Java system	trampoline	exit-only
JAKLD/C [11]	C	a stack of root addresses	trampoline	exit-only
JAKLD/XC [18]	XC-cube (extended C)	scanning roots with nested functions	creating a space-efficient first-class continuation and immediately invoking the continuation	first-class continuation captured with nested functions
JAKLD/SC (this study)	SC-NF (extended SC)	same as previous in column	same as previous in column	same as previous in column

operations over) an auxiliary stack whose elements are addresses of GC roots.

L-closures are lightweight lexical closures created from nested function definitions. In our previous work, we have implemented L-closures by enhancing C compilers [15], [16] and by developing translators from an extended S-expression based C language into the standard C language [9], [14]. For transforming programs written in (extended) S-expression based C languages (called SC languages), we can employ the SC language system [10].

As another LESA mechanism, *closures* are standard lexical closures also created from nested function definitions. We have implemented closures by enhancing C compilers [14], [15], [16]. We can implement closures by developing translators; in fact, we have newly developed such a translator in this study.

In this study, we also reimplemented JAKLD/XC as "JAKLD/SC" written in an extended SC language featuring nested functions (called the SC-NF language in this paper). With JAKLD/SC, we evaluated not only enhanced-C-compiler based LESA mechanisms but also transformation-based portable LESA mechanisms.

Transformation-based implementations of LESA mechanisms are preferable over enhanced C-compiler based implementations in terms of portability and development costs. We developed the extended C compiler [15] by enhancing GCC 3.4.6 at that time. After GCC 4, the inside of the GCC compiler was significantly changed, which made it difficult to enhance GCC for L-closures. As a background for this study, these facts lead to a development plan where we take transformation-based implementations or at least combine them with relatively simple compiler enhancement. As implementations of an LESA mechanism *L-closures* based on translators from SC-NF into standard C, we have developed an implementation which reduces creation/maintenance costs of L-closures [9] which was published in 2006 and called the "execution stack reconstruction technique." Another implementation we developed which reduces creation/maintenance costs plus invocation costs of L-closures [14] was published in 2013 and called the "frame-by-frame restoration technique." Note that the new transformation-based implementation of an LESA mechanism *closures* is realized as a translator into standard C by following the main idea of the compilation techniques for closures in enhancing GCC 4 [14].

In addition to an intermediate report on JAKLD/SC presented in a workshop [17], this paper includes the new transformation-based implementation of closures for evaluating various implementations of LESA mechanisms.

The main contribution of this paper is five-fold.

- We developed a new implementation of an LESA mechanism *closures* based on the translation from an extended SC language that features nested functions (SC-NF) into standard C.
- We reimplemented JAKLD/XC proposed in Ref. [18] as a new Scheme implementation JAKLD/SC written in the SC-NF language. Through the reimplementation, we showed that their difference in syntax made few troubles.
- With JAKLD/SC, we got results of performance measurements on transformation-based implementations of LESA mechanisms (namely, execution stack reconstruction technique [9], frame-by-frame restoration technique [14], and a new transformation-based implementation of *closures*) as well as enhanced C compilers, where the new transformation-based implementation of *closures* showed higher performance than other LESA mechanisms.
- We improved the transformation-based implementations of L-closures by discovering some bugs in the implementations and fixing them.
- Along with the evaluation, we discovered future work and future directions for improving LESA mechanisms.

The remainder of this paper is organized as follows. In Section 2, we describe enhanced-C-compiler based implementations of LESA mechanisms (L-closures and closures). In Section 3, we describe the Scheme language specification, and describe existing Scheme interpreters, JAKLD (written in Java), JAKLD/C (reimplemented in C), and JAKLD/XC (written in an extended C language XC-cube that features L-closures and closures). In Section 4, we describe the extended SC language SC-NF in which JAKLD/SC is written. We also describe existing transformation-based implementations of LESA mechanism *L-closures* using the SC language system. In this section we finally describe a new transformation-based implementation of LESA mechanism *closures* using the SC language system. In Section 5, we describe the (re-)implementation of Scheme interpreter JAKLD/SC written in the SC-NF language. In Section 6, we present discovered bugs in existing transformation-based implementations of L-closures, and we address their fixes. We evaluate JAKLD/SC in Section 7 and describe related work in Section 8. We conclude this paper with future work in Section 9.

Since various language mechanisms, languages, and implementations appear in this paper, we summarize their relationship in **Tables 1**, **2** and **3**. Please see them as well as Table 4 in Section 4.2 if necessary.

Table 2 The nested function-based language mechanisms appearing in this paper and their implementation strategies/techniques.

Language mechanism	Language	Implementation policy	Implementation technique
trampolines	C w/ GCC extensions	compatibility with top level functions.	implemented in the GCC compiler. A function pointer points to an instruction sequence called trampoline which is dynamically generated and stored in the execution stack [3].
closures	XC-cube (extended C)	moderate creation and maintenance costs and equivalent invocation costs to top level functions.	implemented in the enhanced C compiler. Variable values are stored in the execution stack. A function pointer points to a pair of an instruction sequence and an environment [14], [15].
closures	SC-NF (extended SC)	moderate creation and maintenance costs and equivalent invocation costs to top level functions.	Local variables accessed by nested functions are translated into fields of a structure that represents a closure's environment.
L-closures	XC-cube (extended C)	reduces creation and maintenance costs at the expense of higher invocation costs. Lower maintenance costs are achieved by not preventing register allocation.	implemented in the enhanced C compiler. Two locations are prepared for each variable and coherence between the locations is lazily kept. Initialization of L-closures is delayed [15], [16].
L-closures	SC-NF (extended SC)	reduces creation and maintenance costs at the expense of higher invocation costs. Lower maintenance cost is achieved by not preventing register allocation. Incurs relatively small delay judgment costs.	implementation based on translation into standard C. Data in the execution stack are evacuated to an explicit stack when an L-closure is called. The execution stack is reconstructed after the execution of the L-closure [9].
L-closures	SC-NF (extended SC)	reduces creation and maintenance costs at the expense of higher invocation costs, but alleviates the increase in the invocation costs. Lower maintenance costs are achieved by not preventing register allocation. Incurs relatively small delay judgment costs.	implementation based on translation into standard C. Data in the execution stack are evacuated to an explicit stack when an L-closure is called. After the execution of the L-closure, the execution stack is restored frame by frame when necessary [14].

Table 3 Combinations that can be evaluated (Yes) due to this study (2018). An asterisk (*) signifies combinations that were evaluated in this study.

Mechanism	Language	Implementation	Scheme interpreter	
			JAKLD/XC [18]	JAKLD/SC (2018)
trampoline [3]	extended C	GCC-4.6.3	Yes	Yes (2018) (*)
closure	XC-cube	enhanced GCC-3.4.6 [15]	Yes [18]	Yes (2018)
closure	XC-cube	enhanced GCC-4.6.3 [14]	Yes	Yes (2018) (*)
closure	SC-NF	translator into C (2018)		Yes (2018) (*)
L-closure	XC-cube	enhanced GCC-3.4.6 [15]	Yes [18]	Yes (2018) (*)
L-closure	SC-NF	translator into C [9] (execution stack reconstruction)		Yes (2018) (*)
L-closure	SC-NF	translator into C [14] (frame-by-frame restoration)		Yes (2018) (*)

2. LESA Mechanisms: L-Closures and Closures

2.1 L-Closures: An LESA Mechanism

As will be described in Section 3.4, JAKLD/XC [18] employs an enhanced-C-compiler based implementation of a safe LESA mechanism called L-closure [15]. For this implementation, we enhanced the GNU C compiler (GCC) 3.4.6 based on a trampoline-based implementation [3] of nested functions (lexical closures). Nested functions are introduced as a GCC-specific extension to C. To make them compatible with ordinary top-level functions, GCC dynamically generates an instruction sequence on its execution stack where the instruction sequence (also called *trampoline*) sets up a static link and jumps. Note that this technique incurs considerable creation costs for flushing the instruction cache.

By using LESA mechanisms, we can manipulate values of variables of callers deeply in execution stacks. Examples of the typical use of L-closures will be shown in Fig. 2 in Section 3.4.

For the implementation of L-closures [15], [16], at that time, we minimized the overhead of normal execution (the common case) by aggressively reducing creation/maintenance costs of L-closures and by accepting higher invocation costs. To reduce creation costs of L-closures, we proposed a technique for delaying the initialization of an L-closure on memory until the L-closure is actually invoked. To reduce maintenance costs of L-closures, we proposed a technique for making variables accessed by an L-closure still remain register allocation candidates during the normal execution and for delaying the saving of private values into stack-memory locations accessible by the L-closure body until the L-closure is actually invoked.

2.2 Closures: An LESA Mechanism More Moderate Than L-Closures

We have also developed an enhanced-C-compiler based implementation of closures (an LESA mechanism more moderate than L-closures) [14], [15]. Closures incur usual creation/maintenance costs, and also they incur only as usual invocation costs as ordinary top-level functions. Based on the trampoline-based implementation [3] of nested functions (introduced as a GCC-specific extension to C and incurring considerable creation costs for compatibility), we have employed (1) a pair of a closure's own en-

vironment and code instead of an instruction sequence (trampoline) and (2) a *type* and calling convention both incompatible with those for an ordinary function. Consequently, we have reduced creation costs by initializing only an environment-code pair (two words).

We have implemented closures by enhancing not only GCC 3.4.6 [15] but also GCC 4.6.3 [14]. After GCC 4, the inside of the GCC compiler was significantly changed, which makes it difficult to enhance GCC. However, it was found that enhancing GCC 4.6.3 for closures was not as difficult as for L-closures.

3. Existing Scheme Interpreters

3.1 The Scheme Language Specification

Implementations of the Scheme language [1], [12], which is a dialect of Lisp, are required to be properly tail-recursive and to support an unbounded number of active (pre-return) tail calls. As was discussed in the previously proposed implementation [18] (called JAKLD/XC), Clinger proposed a formal definition of *proper tail recursion* for a subset of Scheme in terms of *space efficiency* [4]. Reference [18] proposes an implementation technique which satisfies this definition in terms of asymptotic space complexities. This definition of proper tail recursion encompasses systematic tail call optimization as well as Baker's implementation of Scheme in the C language with CPS (continuation-passing style) conversion [2]. The tail call optimization (for space) is a widely used implementation technique for implementing tail calls, where every tail call is converted to a jump (effectively with an optional trampoline [13]). Note that the target of an optimized call (a jump) may be another procedure, where a *jump* is (1) to finish the current call (i.e., to discard the current environment) before its return then (2) to start a new call by passing the common responsibility of returning (the very current continuation) as well as new actual (evaluated) arguments.

In addition, Scheme implementations are required to support GC and *first-class continuations*.

3.2 JAKLD: A Scheme Interpreter in Java

JAKLD [19] is "a Lisp driver to be embedded in Java applications". Although, according to Ref. [19], JAKLD is designed and developed to be embedded in Java applications, it can serve as a standalone Lisp system.

According to Ref. [19], the key design goals of JAKLD are as follows:

(1) It should be easy for Java programmers to add, delete and modify functionality without previous experience implementing Lisp systems.
(2) It should be easy to implement functionality for dealing with software components written in Java.
(3) The implementation of JAKLD should be compact.
(4) The minimum debugging facilities should be provided, though there is no need to provide powerful development tools for Lisp programming.
(5) The performance should be comparable, though not excellent.

Especially when adding functionality, Scheme procedures and special forms can be directly and easily implemented since JAKLD is not a compiler. Evaluating subexpressions can be implemented simply by invoking `eval` with arguments such as the current environment.

```
1  #define DDl_if def_sp(l_if, "if", "OO", "O", false)
2  static void *l_if(Object cond, Object e1,
3                    Object e2, Env env)
4  {
5    void *result;
6
7    //For scanning GC roots, push addresses of variables
8    //(&e1, &e2, &env) used after call
9    new_push3(e1, e2, env);
10   result = eval(cond, env);
11   new_pop(3); //pop three elements from the root-address stack
12
13   if (!EQP(result, F))
14     return eval(e1, env);
15   else
16     //address-taken variables such as e2 are located on memory
17     //not on registers
18     return (e2 == NULL) ? Nil : eval(e2, env);
19 }
```

Fig. 1 Implementing "`if`" for JAKLD/C.

JAKLD employs a nearly full set of the IEEE Scheme standard. The functionality that does not conform to the IEEE standard includes continuations created by `call/cc` as Scheme procedures. Continuations cannot be invoked after `call/cc` returns, i.e., they are escape procedures. Thus, they can be used for performing non-local exits like catch and throw in Common Lisp but they cannot be used for implementing coroutines.

Although the original version of JAKLD is not properly tail-recursive, a later version supports proper tail recursion by using trampolines [20].

3.3 JAKLD/C: A Scheme Interpreter Reimplemented in C

In this study, we call a (re)implementation of JAKLD in C [11] JAKLD/C. JAKLD/C is based on the language specification for JAKLD without supporting so-called bignums. It consists of an interpreter and a memory manager. It features a customizable memory manager so that it can serve as a GC research platform. Since all GC algorithms must be able to scan GC roots and scan objects, necessary code has been implemented as a common part. The system manages parameters and local variables as roots by using a stack whose elements are addresses of roots. Unfortunately, this scheme prevents such variables from being register-allocated.

For example, Scheme's `if` is *directly* implemented, as shown in **Fig. 1**. Note that JAKLD/C uses `l_if` as a C function name since `if` is a reserved keyword in C. JAKLD/C makes its code compact by inheriting the advantages of JAKLD. For scanning GC roots, it employs a macro for taking addresses of roots and pushing them onto a stack whose elements are addresses of roots.

3.4 JAKLD/XC: Implemented in an Extended C Language That Features L-Closures (or Closures)

JAKLD/XC [18] employs L-closures [15], [16] for improving performance about GC. For example, Scheme's `if` is implemented, as shown in **Fig. 2**. By adding keyword `lightweight` to nested function `scan1`, we can create an L-closure by evaluating the nested function definition. To implement copying GC, by `e1 = mv(e1)`, we move a live object pointed to by root `e1` into a *To*-space of the *heap*. In this code, L-closure `scan1` calls an L-closure `scan0` to scan roots in the caller of `l_if`; this process is repeated (e.g., for the caller of the caller) until all roots

```
static void *l_if(scanL scan0, Object cond,
                  Object e1, Object e2, Env env)
{
  void *result;

  //For scanning GC roots, create L-closure scan1
  void scan1 lightweight (move_f mv) {
    scan0(mv); //scan0 scans roots in the caller of l_if
    //variables such as e1 may be located on registers during the
    //normal execution
    e1 = mv(e1); e2 = mv(e2); env = mv(env);
  }

  result = eval(scan1, cond, env);

  if (!EQP(result, F))
    return eval(scan0, e1, env);
  else
    return (e2==NULL) ? Nil : eval(scan0, e2, env);
}
```

Fig. 2 Implementing copying GC with L-closures on implementing "`if`" for JAKLD/XC.

```
/* globals */
scanL scan_in_capture;
void *last_val;
Frame x_frame;

static void *l_if(scanL scan0, Object cond,
                  Object e1, Object e2, Env env) {
 void *result;
 int pc = 0;
 void scan1 lightweight (enum why_scan w, move_f mv)
 {
  switch(w) {
  case ForGC:
   scan0(w, mv);
   e1 = mv(e1); e2 = mv(e2); env = mv(env);
   break;
  case ForCapture:
   scan0(w, mv);
   SAVE_FRAME(x_frame, scan_in_capture, l_if_c, pc);
   SAVE_PARAM4(x_frame, scan_in_capture,
               NULL, e1, e2, env);
   SAVE_LOCAL0(x_frame, scan_in_capture);
  }
 }

 if (last_val) {
  pc = REF(x_frame, pc); RESTORE_LOCAL0(x_frame);
  x_frame = 0;
  switch(pc) {
  case 1: result = last_val; last_val = 0; goto L1;
  }
 }

 pc = 1;
 result = eval(scan1, cond, env);
 L1:;

 if (!EQP(result, F))
  return eval(scan0, e1, env); // tail call
 else
  return (e2==NULL) ? Nil
                    : eval(scan0,e2,env); // tail call
}

static void l_if_c(scanL scan0) {
 Vector params = REF(x_frame, params);
 last_val = l_if(scan0, REF(params, value[0]),
                        REF(params, value[1]),
                        REF(params, value[2]),
                        REF(params, value[3]));
}
```

Fig. 3 Implementing copying GC and first-class (captured) continuations with L-closures on implementing "`if`" for JAKLD/XC.

(parameters/local variables) in the entire C stack are scanned.

The enhanced-C-compiler based implementations of L-closures [15], [16] attempt not to prevent variables `e1`, `e2`, `env` from being register-allocated. We prepare two locations for each of these variables: one private location is a register allocation candidate, and another shared location is put on stack memory. operations for saving private values into shared locations are delayed until an L-closure is actually invoked. During the invocation of the L-closure, it accesses the shared locations. On returning to the function that owns the variables, private values are restored from shared locations. To reduce creation costs of L-closures, Operations for initializing L-closures are also delayed until an L-closure is actually invoked.

By using keyword `closure` in place of `lightweight` for nested function `scan1`, we can create a *closure* by evaluating the nested function definition.

JAKLD/XC [18] implements proper tail recursion based on the key idea that is to "create a space-efficient first-class continuation and immediately invoke it". JAKLD/XC performs this for resetting the use of the execution stack every time execution stack is about to overflow based on a criterion. By creating a space-efficient first-class continuation, JAKLD/XC improves space efficiency. By using a criterion that the stack space is less than some fixed constant, the space consumption for an execution stack is actually less than some fixed constant. When evaluating asymptotic space complexities according to Clinger's formal definition, we can ignore constant terms in space consumption functions. Therefore, we can confirm that JAKLD/XC is a properly tail-recursive implementation.

Scheme's `if` is implemented for not only scanning GC roots but also capturing continuations, as shown in **Fig. 3**. As an auxiliary function to execute the continuation captured as a list of "Frame" objects, function `l_if_c` is added. In this implementation, variable `last_val` represents a result of the last evaluation, and variable `x_frame` represents a frame being captured or a frame to be executed next; both are employed like machine registers. Variable `scan_in_capture` is used for scanning GC roots even when GC is started during the capturing of continuations. For more details, please refer to the paper [18].

4. Extended SC Languages with Nested Functions

4.1 SC Language System

We can reduce development cost for transformation-based implementations of language extensions by using the S-expression based C language (SC language) system [10].

The SC language is a generic term for the C language with S-expression based syntax and its extended languages. Among them, the SC language with no extension is called the SC-0 language. The SC language system provides a translator from the SC-0 language to the C language. Thus, users can complete implementations of extended SC languages by implementing translators to SC-0.

The SC language system consists of the following three kinds of modules.

SC preprocessors
: perform preprocessing such as inserting the contents of files specified by `%include` directives and macro expansion (as preprocessors of the C language),

SC translators
: take an S-expression (source code) and a transformation ruleset as inputs and transform the S-expression by applying the rules, and

The SC compiler
: compiles SC-0 code into C.

Our implementation of the SC language system runs on Common Lisp.

4.2 SC-NF: An Extended SC Language with Nested Functions

The SC-NF language, an extended SC language with nested functions, can be translated into the C language with GCC extensions (including nested functions), the XC-cube language with

Table 4 SC-NF's translation target languages by the SC language system.

Source language	Target language	Intermediate language	Remarks
SC-NF	C w/ GCC extensions (incl. nested functions)	SC-0-gcc	
SC-NF	XC-cube (L-closures)	SXC-0 (L-closures)	
SC-NF	XC-cube (closures)	SXC-0 (closures)	
SC-NF	Standard C (w/o nested functions)	SC-0 (w/o nested functions)	CL-SC2: closures based on environment-to-structure conversion (2018)
SC-NF	Standard C (w/o nested functions)	SC-0 (w/o nested functions)	LW-SC: L-closures based on the execution stack reconstruction technique [9]
SC-NF	Standard C (w/o nested functions)	SC-0 (w/o nested functions)	LW-SC2: L-closures based on the frame-by-frame restoration technique [14]

L-closures, the XC-cube language with closures, or the standard C language (without nested functions), using the SC language system. **Table 4** summarizes the translation targets of the SC language system. Here, translation into standard C is used to implement L-closures or closures.

The following code should be written at the beginning of an SC-NF program.

```
1 (%ifndef* NF-TYPE
2   ;; one of (GCC XCC XCCCL CL-SC2 LW-SC)
3   (%defconstant NF-TYPE GCC))
4 (%include "rule/nestfunc-setrule.sh")
```

Here, one of `GCC`, `XCC`, `XCCCL`, `CL-SC2` and `LW-SC` is chosen as the `NF-TYPE`. The program is translated into the C language with GCC extensions (including nested functions), the XC-cube language with L-closures, the XC-cube language with closures, standard C that implements closures, or standard C that implements L-closures based on the execution stack reconstruction technique [9], respectively. All of these translations are implemented with the `nestfunc` ruleset, a transformation ruleset of the SC language system.

In our current implementation, programmers need to use another version of the ruleset to use the translation into standard C that implements L-closures based on the frame-by-frame restoration technique [14]. We will enhance the `nestfunc` ruleset to enable programmers to choose this implementation only by specifying `LW-SC2` as the `NF-TYPE`. Thus, we denote the translation into the standard C that implements L-closures based on the frame-by-frame restoration technique [14] as LW-SC2 in this paper (especially in the figures and tables).

In SC languages, the type of a function is written separately from its parameter list as `(fn` *ret-type parameter-type*$\cdots$`)`. For example, see lines 1–2 and 34 in Fig. 4 in Section 5. In the SC-NF language, the type of a nested function is written as `(NESTFN` *ret-type parameter-type*$\cdots$`)`. For example, see line 6 in Fig. 4.

4.3 Translation Techniques for Implementing L-Closures

The translator from the SC-NF language into the standard C language (via the SC-0 language) that implements L-closures based on execution stack reconstruction or frame-by-frame restoration has been implemented as a transformation ruleset. Note that we uniformly call the extended SC language with nested functions as the SC-NF language in this paper. In some literature such as Refs. [9], [14], the extended language with nested functions whose types are written as `(lightweight` *ret-type parameter-type*$\cdots$`)` is refered to as the LW-SC language.

The execution stack reconstruction technique [9]
reduces the creation and maintenance costs of L-closures by delaying data evacuation to an explicit stack until an L-closure is called to promote the use of the execution stack. A drawback of this technique is that the overall performance is degraded when nested functions are called frequently. This degradation is brought by overhead for each nested function call. The overhead comprises the costs of evacuating necessary data in the C execution stack into the explicit stack to enable the data to be accessible by the nested function and reconstructing the C execution stack after the execution of the nested function to resume the execution from its caller. The latter cost is especially considerable.

The frame-by-frame restoration technique [14]
is implemented based on the execution stack reconstruction technique with the improvement of reducing the invocation cost of nested functions. In the execution stack reconstruction technique, when the execution of a nested function completes, the C execution stack is reconstructed using all the data evacuated to the explicit stack. In the frame-by-frame restoration technique, the C execution stack is restored frame by frame when necessary. Thus, we can reduce the invocation costs of nested functions because the amount of data transferred between the C and explicit stacks is reduced when a nested function is called again.

The performance evaluation section (Section 7) reveals that the *delay judgment costs* are also considerable in these transformation-based implementations of L-closures. The delay judgment costs are necessary in addition to the creation, maintenance, and invocation costs of L-closures described above. More detailed discussion is presented in Section 7 referring to Fig. 7 and Fig. 8 in Section 6. This kind of cost has been overlooked maybe because L-closures and closures based on C compiler enhancements (XC-cube compilers) are carefully implemented to avoid such cost.

4.4 Translation Techniques for Implementing Closures

In this study, we implemented a new translator, as a new transformation ruleset, from the SC-NF language into the standard C language (via SC-0) that implements closures. This implementation does not incur the delay judgment costs mentioned in the previous section.

This translator into standard C follows the same basic idea as the closure implementation proposed in Ref. [14], which is implemented as an enhanced GCC 4 compiler. Concretely, it translates

local variables accessed by nested functions into fields of a structure representing a closure's environment. Unlike the translators for L-closures, which translate programs to evacuate/restore the data of all the local variables to/from the explicit stack, this translator for closures leaves local variables that are not accessed by nested functions untreated. In addition, declarations of local variables that are translated into fields (except for function parameters) are removed. A translation example is shown in Section 6.

The new implementations based on translation into standard C are more portable than the existing implementations as enhanced C compilers. In addition, the new implementations may achieve better performance due to optimizations made by back-end C compilers.

An SC language system user can use this implementation by choosing `CL-SC2` as `NF-TYPE`. This name comes from `CL-SC`, the name of the previous version of the transformation-based implementation of closures. `CL-SC` was implemented by modifying `LW-SC`, the transformation-based implementation of L-closures, to use an explicit stack without delay.

5. JAKLD/SC: A Scheme Interpreter written in SC-NF

Figure 4 shows an example of reimplementation in SC-NF corresponding to Fig. 3 written in XC-cube. (Some details are omitted.) As you can see, this reimplementation is basically to change only syntax. Indeed, we found no problem in most C code in rewriting it into the SC language. However, we found a few very concrete problems arising from the syntax difference. This paper presents them to show their impacts and how to address them below.

For example, macros `REF` and `UPDATE_P_INI` in **Fig. 5** had problems. `REF(obj, field)` is used for accessing an object `obj`'s field `field`. `UPDATE_P_INI` is used for initializing an object `obj`'s field `field` with pointer `val`. In JAKLD/C, we can implement GC algorithms with read barriers and/or write barriers by redefining these macros, if necessary, according to the GC algorithms. In C, we can write `UPDATE_P_INI(v, value[i], fill)` since we can write `REF(v, value[i])` in place of `REF(v, value)[i]`. However, this relies on the fact that, after expanding `REF(v, value[i])`, `v->value[i]` is parsed as `(v->value)[i]`. Since, after expanding `REF(v, (value[i]))`, `v->(value[i])` causes a syntax error even in C, the same trick does not work in SC. In SC, we need to define two separated macros for `REF(v, value)` and `REF(v, value[i])`, and to define `UPDATE-ELM-P-INI` in addition to `REF`-based `UPDATE_P_INI`, as shown in **Fig. 6**.

Usage of character literals were also discovered as having a problem. In SC, character literals are those in Lisp; thus, we need to translate character literals in C such as `'\036'` into the character literals in Lisp whose character codes are the octal numbers. Sometimes Lisp does not provide such literals. To solve this problem, we define a new macro as `(%defmacro c-char (x) (code-char x))`, then we write `(c-char #o36)`. In a macro definition, we usually employ backquote macros (pseudo quotations) as in Fig. 6. Instead of them, we directly use an expression in Common Lisp. This implies an evaluation of an expression in Common Lisp; thus, we should limit expressions for macros in future work.

```
1  (static (l_if scan0 cond e1 e2 env)
2        (fn (ptr void) scanL Object Object Object Env)
3   (def result (ptr void))
4   (def pc int 0)
5   (def (scan1 w mv)
6        (NESTFN void (enum why-scan) move-f)
7     (switch w
8       (case ForGC)
9       (scan0 w mv)
10      (= e1 (mv e1)) (= e2 (mv e2)) (= env (mv env))
11      (break)
12      (case ForCapture)
13      (scan0 w mv) ... ))
14
15  (if last_val
16      (begin
17        (= pc (REF x_frame pc))
18        (= x_frame 0)
19        (switch pc
20          (case 1)
21          (= result last_val) (= last_val 0)
22          (goto L1))))
23
24  (= pc 1)
25  (= result (eval scan1 cond env))
26  (label L1 ())
27
28  (if (not (EQP result F))
29      (return (eval scan0 e1 env))
30    (return (if-exp (== e2 NULL)
31                    Nil
32                    (eval scan0 e2 env)))))
33
34 (static (l_if_c scan0) (fn void scanL)
35  (def params Vector (REF x_frame params))
36  (= last_val (l_if scan0 (REF-ELM params value 0)
37                          (REF-ELM params value 1)
38                          (REF-ELM params value 2)
39                          (REF-ELM params value 3))))
```

Fig. 4 Implementing copying GC and first-class (captured) continuations with L-closures on implementing "`if`" for JAKLD/SC. (Some details are omitted.)

```
1 #ifndef REF
2 #define REF(obj, field) obj->field
3 #endif
4 #ifndef UPDATE_P_INI
5 #define UPDATE_P_INI(obj, field, val) \
6         REF(obj, field) = val
7 #endif
```

Fig. 5 Macro definitions in C for the existing implementation.

```
1  (%ifndef REF
2  ((%defmacro REF (obj field)
3      `(fref (mref ,obj) ,field))))
4  (%ifndef REF-ELM
5  ((%defmacro REF-ELM (obj a i)
6      `(aref (fref (mref ,obj) ,a) ,i))))
7  (%ifndef UPDATE-P-INI
8  ((%defmacro UPDATE-P-INI (obj field val)
9      `(= (REF ,obj ,field) ,val))))
10 (%ifndef UPDATE-ELM-P-INI
11 ((%defmacro UPDATE-ELM-P-INI (obj a i val)
12     `(= (REF-ELM ,obj ,a ,i) ,val))))
```

Fig. 6 Macro definitions in SC.

6. Problems in Transformation-Based Implementations of LESA Mechanisms and Their Solutions

In this study, we developed the Scheme implementation JAKLD/SC, which is written in an extended SC language SC-NF, so that JAKLD can use implementations of LESA mechanisms based on translation into the standard C language. Through this development, we found out some interesting problems and solved them.

Figure 7 and **Fig. 8** show the translation results of the function `l_if_c` in lines 34–39 of Fig. 4 based on execution stack reconstruction [9] and frame-by-frame restoration [14] techniques, respectively. See Refs. [9] and [14] for details of these techniques, respectively.

Note that the code in these figures shows the results after the problems are solved. Before solving the problems, the return val-

```
1  struct l_if_c_frame
2  {
3   char *tmp_esp;
4   char *argp;
5   int call_id;
6   struct Vector *params;
7   closure_t *scan0;
8  };
9
10 void l_if_c (char *esp, closure_t * scan0)
11 {
12  void *nf_retval_tmp118;
13  struct Vector *params;
14  size_t esp_flag = (size_t) esp & 3;
15  char *new_esp;
16  struct l_if_c_frame *efp;
17  if (esp_flag)
18   {
19    esp = (char *) ((size_t) esp ^ esp_flag);
20    efp = (struct l_if_c_frame *) esp;
21    esp =
22      (char *) ((Align_t *) esp +
23                (sizeof (struct l_if_c_frame)
24                 + sizeof (Align_t) + -1)
25                / sizeof (Align_t));
26    *((char **) esp) = 0;
27   LGOTO:switch ((*efp).call_id)
28      {
29      case 0:
30        goto L_CALL120;
31      }
32    goto L_CALL120;
33   }
34  efp = (struct l_if_c_frame *) esp;
35  esp =
36   (char *) ((Align_t *) esp +
37             (sizeof (struct l_if_c_frame)
38              + sizeof (Align_t) + -1)
39             / sizeof (Align_t));
40  *((char **) esp) = 0;
41  params = (*x_frame).params;
42  {
43   new_esp = esp;
44   while (__builtin_expect
45          ((nf_retval_tmp118 =
46            l_if (new_esp, scan0,
47                  ((*params).value)[0],
48                  ((*params).value)[1],
49                  ((*params).value)[2],
50                  ((*params).value)[3]))
51           == (void *) ((long) -1),
52           0)
53         && __builtin_expect
54            ((efp->tmp_esp = *((char **) esp)) != 0,
55             1))
56    {
57      efp->params = params;
58      efp->scan0 = scan0;
59      efp->call_id = 0;
60      return;
61    L_CALL120:;
62      params = efp->params;
63      scan0 = efp->scan0;
64      new_esp = esp + 1;
65    }
66   last_val = nf_retval_tmp118;
67  }
68 }
```

Fig. 7 The translation result of the `l_if_c` function in Fig. 4 on implementing L-closures based on the execution stack reconstruction technique (LW-SC).

ues of the `l_if` function called in line 46 of Fig. 7 and line 36 of Fig. 8 are directly assigned to `last_val`. In LW-SC, the integer -1 is used as a special function return value to indicate that this return is not a normal function return but a temporary return to prepare for a nested function call by unwinding the C execution stack. Therefore, the function return value should not be assigned to `last_val` until this function return turns out to be a normal one. In the JAKLD/SC case, the garbage collector did not work correctly because `last_val` is accessed by nested functions during root scanning.

Another less significant problem in the execution stack reconstruction technique is that the complex expressions appear in the argument list of the `l_if` function call. In this technique, this function is called for every reconstruction of the C stack and thus these expressions are evaluated every time. In example Fig. 7, the complex expressions such as `((*params).value)[0]` appear in the argument list. This seems not to be a problem since these expressions do not have any side effects, but, in environments where copying GC is used as in this study's Scheme language system, it is possible that the object pointed to by `params` has been garbage collected and `params` points to invalid data when these expressions are reevaluated. However, this problem does not surface because such invalid data are overwritten by evacuated values. For example, the parameter `scan0` in line 10 of Fig. 7 is overwritten by an evacuated value in line 63. In the same way, the invalid parameter values of `l_if` in lines 46–50 are also overwritten in the callee when this function is called for reconstruction of the execution stack. (Note that the LSB of `new_esp` is used for indicating whether the function is called normally or called for reconstruction of the execution stack.) Nonetheless, to be strict, the translated code should store the result values of the argument expressions in temporary variables. In addition, the increase in the number of such temporary variables should be prevented as much as possible using techniques such as reusing a temporary variable in other function call expressions.

```
1  struct l_if_c_frame {
2    struct fframe *parent_fr;
3    struct fframe *child_fr;
4    struct fframe *xfp;
5    fn_or_clos_t f_or_c;
6    int call_id;
7    closure_t *scan0;
8    struct Vector *params;
9  };
10
11 void l_if_c (char *esp, closure_t * scan0) {
12   void *nf_retval_tmp118;
13   struct Vector *params;
14   size_t esp_flag = (size_t) esp & 3;
15   struct l_if_c_frame *efp;
16   if (esp_flag)
17     {
18       esp = (char *) ((size_t) esp ^ esp_flag);
19       efp = (struct l_if_c_frame *) esp;
20       esp = (char *) ((Align_t *) esp +
21               (sizeof (struct l_if_c_frame)
22                + sizeof (Align_t) + -1)
23               / sizeof (Align_t));
24     LGOTO: switch ((*efp).call_id)
25              { case 1: goto L_CALL141; }
26       goto L_CALL141;
27     }
28   efp = (struct l_if_c_frame *) esp;
29   esp = (char *) ((Align_t *) esp +
30                   (sizeof (struct l_if_c_frame)
31                    + sizeof (Align_t) + -1)
32                   / sizeof (Align_t));
33   params = (*x_frame).params;
34   if (__builtin_expect
35       ((nf_retval_tmp118 =
36         l_if (esp, scan0,
37                 ((*params).value)[0],
38                 ((*params).value)[1],
39                 ((*params).value)[2],
40                 ((*params).value)[3]))
41        == (void *) ((long) -1),
42       0)
43       && __builtin_expect
44          (((struct fframe *) esp)->child_fr,
45           1))
46     {
47       efp->params = params;
48       efp->scan0 = scan0;
49       (efp->f_or_c).fn = (normfn_t) l_if_c_comp;
50       ((struct fframe *) esp)->parent_fr =
51         (struct fframe *) efp;
52       efp->child_fr = (struct fframe *) esp;
53       efp->call_id = 1;
54       return;
55     L_CALL141:;
56       params = efp->params;
57       scan0 = efp->scan0;
58       struct child_frame {
59         struct fframe *parent_fr;
60         struct fframe *child_fr;
61         struct fframe *xfp;
62         fn_or_clos_t f_or_c;
63         int call_id;
64         void *retval;
65       };
66       struct child_frame *cfp;
67       cfp = (struct child_frame *) efp->child_fr;
68       last_val = cfp->retval;
69     }
70   else
71     last_val = nf_retval_tmp118;
72   efp->child_fr = 0;
73 }
74
75 int l_if_c_comp (struct l_if_c_frame *efp) {
76   l_if_c ((char *) efp + 1, efp->scan0);
77   if (efp->child_fr) return 1; else; return 0;
78 }
```

Fig. 8 The translation result of the `l_if_c` function in Fig. 4 on implementing L-closures based on the frame-by-frame restoration technique (LW-SC2).

Code resulting from translation using the frame-by-frame restoration technique [14] includes the additional auxiliary function `l_if_c_comp`. Note that such auxiliary functions are de-

fined at the C language level for *implementing* SC-NF's nested functions, while, in the JAKLD/SC implementation, we define auxiliary functions such as `l_if_c` in SC-NF for executing continuations captured by *using* SC-NF's nested functions.

In this study, we developed a new transformation-based implementation of closures to avoid the delay judgment costs, which were found to be a problem in the performance evaluations presented in Section 7. **Figure 9** shows the translation result of the function `l_if_c` in lines 34–49 of Fig. 4. We can see that

```
1  struct l_if_c_frame {};
2
3  void l_if_c (closure_t * scan0) {
4    struct Vector *params;
5    struct l_if_c_frame my_frame;
6    struct l_if_c_frame *efp;
7    efp = &my_frame;
8    params = (*x_frame).params;
9    last_val = l_if (scan0, ((*params).value)[0],
10                           ((*params).value)[1],
11                           ((*params).value)[2],
12                           ((*params).value)[3]);
13 }
```

Fig. 9 The translation result of the `l_if_c` function in Fig. 4 on implementing closures based on environment-to-structure conversion (CL-SC2).

```
1  struct scan1_in_l_if_frame {};
2
3  struct l_if_frame {
4   int pc; struct Env *env;
5   union Object *e2; union Object *e1;
6   union Object *cond; closure_t *scan0;
7  };
8
9  void scan1_in_l_if (void *xfp0, enum why_scan w,
10                      void *(*mv) (void *)) {
11  ...
12   struct scan1_in_l_if_frame my_frame;
13   struct scan1_in_l_if_frame *efp;
14   struct l_if_frame *xfp = xfp0;
15   efp = &my_frame;
16   switch (w)
17    {
18    case ForGC:
19     ((void(*)(void*,enum why_scan,void*(*)(void*)))
20     (xfp->scan0)->fun) ((xfp->scan0)->fr, w, mv);
21     xfp->e1 = mv (xfp->e1); xfp->e2 = mv (xfp->e2);
22     xfp->env = mv (xfp->env);
23     break;
24    case ForCapture:
25     ((void(*)(void*,enum why_scan,void*(*)(void*)))
26     (xfp->scan0)->fun) ((xfp->scan0)->fr, w, mv);
27    ...
28    }
29 }
30
31 static void*
32 l_if (closure_t * scan0, union Object *cond,
33       union Object *e1, union Object *e2,
34       struct Env *env) {
35  void *tmp24; struct Pair *ifexp_result40;
36  void *result;
37  closure_t scan1;
38  struct l_if_frame my_frame; struct l_if_frame *efp;
39  efp = &my_frame;
40  efp->scan0 = scan0; efp->cond = cond;
41  efp->e1 = e1; efp->e2 = e2; efp->env = env;
42  scan1.fun = (nestfn_t) scan1_in_l_if;
43  scan1.fr = (void *) efp;
44  efp->pc = 0;
45  if (last_val)
46   {
47    efp->pc = (*x_frame).pc;
48    x_frame = 0;
49    switch (efp->pc)
50      {
51      case 1:
52        result = last_val; last_val = 0;
53        goto L1;
54      }
55   }
56  else;
57  efp->pc = 1;
58  result = eval (&scan1, efp->cond, efp->env);
59 L1:;
60  if (!((void *) result == (void *) F))
61   {
62    tmp24 = eval (efp->scan0, efp->e1, efp->env);
63    return tmp24;
64   }
65  else
66   {
67    if (efp->e2 == NULL)
68      ifexp_result40 = Nil;
69    else
70      ifexp_result40 =
71        eval (efp->scan0, efp->e2, efp->env);
72    return ifexp_result40;
73   }
74 }
```

Fig. 10 The translation result of the `l_if` function in Fig. 4 on implementing closures based on environment-to-structure conversion. (CL-SC2) (Some details are omitted.)

the translation result of a function that does not have nested functions is almost identical to the original one. In fact, the statement `efp = &my_frame;`, and the variables `my_frame` and `efp`, whose types include `struct l_if_c_frame`, are optimized away with compilation into machine code. In addition, **Fig. 10** shows the translation result from the function `l_if` in Fig. 4. As in this example, a local variable accessed by nested functions is translated into a field of a structure, and a nested function is translated into a top-level function that takes a pointer to such a structure object as an additional parameter. When calling a closure, a pointer to the structure object should be passed as the additional argument, as done in lines 19–20 of Fig. 10.

7. Performance Evaluation

In this section, we evaluate performance, mainly execution performance. **Table 5** shows our evaluation environment. We measure the performance using the Gabriel benchmark [7].

- boyer: a benchmark based on Boyer's theorem prover, it includes many `cons` operations
- fft: fast Fourier transform, it includes many floating point operations and array references
- tak: a variant of the Takeuchi function, it includes many recursive calls
- traverse: a benchmark that creates and traverses a tree structure
- cpstak: a CPS (continuation passing style) version of tak
- puzzle: a program that searches for all solutions of a puzzle. First-class continuations by `call/cc` are used for non-local exits from the main search loop.

We also use the following programs:

- fib: a program that calculates the 30th Fibonacci number recursively
- fib-cps: fib in CPS

For each JAKLD/SC implementation, we employ two 50 MB spaces as the heap for Cheney-style copying GC. We limit the number of nested SC-NF calls for interpreting Scheme procedures/constructs to 1,500. If the interpreter exceeds the limit, it creates and invokes a first-class continuation for preventing stack overflow. As shown in Fig. 4, LESA mechanisms are called when roots are scanned or first-class continuations are created (capturing continuations). Roots are scanned infrequently because this occurs only when the current 50 MB space is full. Note that a 50 MB space is consumed, for example, by `cons` in Scheme programs and creation of environments for the Scheme interpreter. First-class continuations are created only for preventing stack overflow except explicit `call/cc`'s. Thus, the creation is frequent if Scheme programs are written in CPS and active (pre-return) tail calls are nested deeply. Otherwise it is infrequent.

Table 5 Evaluation environment.

	UltraSPARC T2 Plus Server
CPU	UltraSPARC T2 Plus 1.4GHz 8-Core×2 8 threads/core (128 threads total)
Memory	24GB
OS	SunOS 5.10
Compiler	L-closure XC-cube (SPARC 32 bit GCC 3.4.6-based [15]) `-g -O2` closure XC-cube (SPARC 32 bit GCC 4.6.3-based [14]) `-g -O2` CL-SC2 (closure) (with SPARC 32 bit GCC 4.6.3 `-g -O2`) LW-SC (L-closure) [9] (with SPARC 32 bit GCC 4.6.3 `-g -O2`) LW-SC2 (L-closure) [14] (with SPARC 32 bit GCC 4.6.3 `-g -O2`) GCC extension (trampoline [3]) (SPARC 32 bit GCC 4.6.3) `-g -O2`

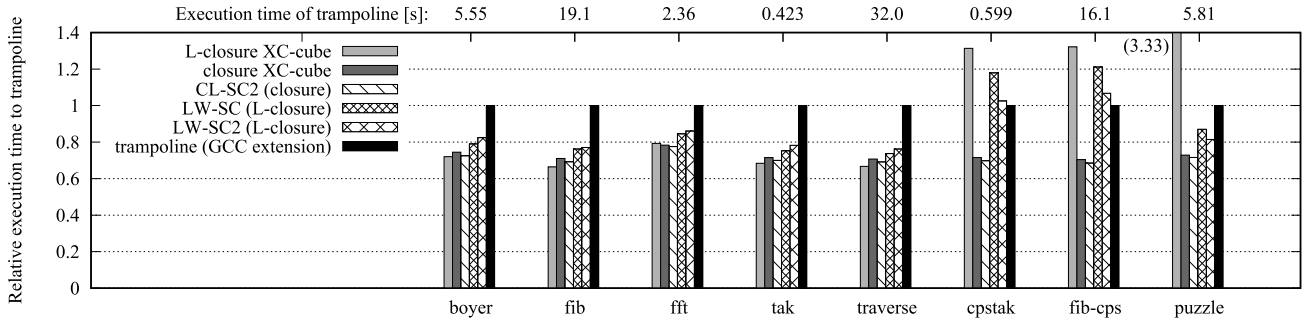

Fig. 11 The measurement results on SPARC. The vertical axis shows relative times to trampoline (GCC extension) (lower is better). The numbers above the graph denote the absolute execution times of the baseline.

Figure 11 shows the measurement results. The vertical axis shows the execution times relative to trampoline (GCC extension), which is mentioned later. The numbers above the graph denote the absolute execution times of trampoline (GCC extension). We compare the following implementations of nested functions, and details of them are shown also in Table 5.

- L-closure XC-cube: L-closures in XC-cube (extended C) implemented by the enhanced C compiler. At the expense of higher invocation costs, other kinds of costs are reduced.
- closure XC-cube: Closures in XC-cube (extended C) implemented by the enhanced C compiler. Invocation costs and creation/maintenance costs are moderate.
- CL-SC2 (closure): Closures implemented based on translation into standard C. Invocation costs and creation/maintenance costs are moderate.
- LW-SC (L-closure): L-closures implemented based on translation into standard C using the execution stack reconstruction technique.
- LW-SC2 (L-closure): L-closures implemented based on translation into standard C using the frame-by-frame restoration technique.
- trampoline (GCC extension): Nested functions in GCC extensions implemented by the GCC compiler. Instruction sequences generated in the C stack (trampolines) are used. This implementation is used as the baseline of relative execution times in Fig. 11 because this can be used easily on many systems only by installing the GCC package. This implementation achieves lower invocation costs, but creation costs are high on SPARC processors due to flushing of the instruction cache.

Among these, what this study enable to be evaluated are implementations based on translation into standard C, namely CL-SC2 (closure), LW-SC (L-closure), and LW-SC2 (L-closure).

With Scheme programs, such as Boyer, fib, fft, tak, and traverse, where the nest of active (pre-return) calls is not so deep and only a limited part of the execution stack is used, transformation-based implementations of L-closures (each based on either the execution stack reconstruction technique or the frame-by-frame restoration technique) achieve 7%∼15% lower performance than enhanced C compilers; this performance would be acceptable when considering transformation-based implementations that do not require enhanced C compilers.

On the other hand, as in the case of programs written in CPS, such as cpstak and fib-cps, with Scheme programs which frequently need to create a space-efficient first-class continuation to support proper tail recursion every time execution stack is about to overflow, transformation-based implementations of L-closures achieve better performance than enhanced C compilers; the frame-by-frame restoration technique is especially effective for reducing the invocation costs as intended. Note that, when a nested function invokes another nested function, most data in the execution stack have already been moved to the explicit stack in LW-SC, whereas, to invoke another nested function as an L-closure in XC-cube, the caller needs to find the nested function to be invoked in the execution stack each time from the top of the stack. This difference appears in the cost of every invocation.

Since **puzzle** uses first-class continuations by `call/cc`, L-closures in XC-cube showed a substantially long execution time. In our previous evaluation [18], by making the C stack empty just after capturing continuations with `call/cc` and leaving only a list of "Frame" objects that represents the continuation, we can promote sharing of continuations and we achieved good performance. Although the implementations based on JAKLD/XC [18] support (non-escape only) first-class continuations, we will discuss first-class continuations in future work.

We also compared transformation-based implementations of L-closures against enhanced-C-compiler based implementations of closures, which is a mechanism more moderate than L-closures. (The comparison to CL-SC2 will be shown later.) Closures incur usual creation/maintenance costs, but they incur only as usual invocation costs as ordinary top-level functions. Although L-closures are implemented by enhancing GCC 3.4.6 and closures are implemented by enhancing GCC 4.6.3, closures show remarkably good performance. Neither transformation-based implementations of L-closures (neither the execution stack reconstruction technique nor the frame-by-frame restoration technique) can achieve better performance than closures. This would be because transformation-based implementations need some *delay judgment costs* and the costs are relatively high in the case of our Scheme interpreter where normal top-level function calls in the implementation language are frequent. Delay judgment costs are required even in (transformed) normal top-level functions for enabling delayed (lazy) save and restore of data on registers (as private locations) into/from the explicit stack (memory) (as shared locations). Note that delay judgment costs were zero in the enhanced-C-compiler based implementation. Concretely

speaking, in Fig. 7 in Section 6, lines 14–40 (code for resuming after forcing delayed save into the explicit stack) are required for using a single function also for stack reconstruction, and, as the delay judgment costs, lines 14, 17, 34–40 are needed to be (always) executed even when the function does not reconstruct the C execution stack. Likewise, in Fig. 8, lines 14, 16, 28–32, 72 are needed to be always executed. When the frame-by-frame restoration technique was proposed in [14], the frame-by-frame restoration technique mostly achieved better performance than closures in the same evaluation environment as Table 5. The application of L-closures used in this study (the Scheme interpreter) showed a different result.

Since transformation-based implementations of L-closures have the problem of delay judgment costs, this study developed a transformation-based implementation of closures (CL-SC2) which does not incur these costs. We expected that CL-SC2 achieved a compatible performance to the enhanced-C-compiler based implementation of closures. In practice, CL-SC2 achieved 1.0%~2.9% better performance than the enhanced-C-compiler based implementation. This may be because the compiler optimizations on compiling from the standard C language are effective. The transformation into standard C achieved not only portability but also better performance than expected.

Let us discuss the code sizes on the implementations of Scheme interpreter JAKLD/SC. To evaluate code size, we use the size of the "`.text`" section that contains instruction sequences in an executable. Unlike lines of (extended) C code, this size excludes comments, unnecessary newlines, structure definitions, and other stuff to be optimized away, and includes instruction sequences added by C compilers. As the standard Scheme implementation, the one with trampoline (GCC extension) has a 69,880-byte code size out of an 89,492-byte `strip`ped executable. We use this for normalizing code size below. The standard implementation contains code to generate trampolines at runtime. The implementation with XC-cube L-closures has approximately 1.35 times code as standard since it contains code for implementing special control flows in case of L-closure calls. LW-SC has approximately 3.99 times code and LW-SC2 has approximately 3.91 times code, since transformation-based implementations of L-closures contain code for using explicit stacks (lazily). The implementation with XC-cube closures has approximately 0.93 times code size since it does not contain code to generate trampolines. The one with the transformation-based implementation of closures (CL-SC2) has approximately 0.95 times code.

8. Related work

The transformation-based implementation of closures in this paper, which translates an SC-NF language into standard C, realizes the same basic idea for implementing closures [14] as a translator instead of modifying GCC 4. This translator can be considered as standard closure conversion except that structures expressing the environment are placed on the stack. LESA mechanisms are designed without requiring garbage collection because implementing garbage collection is one of the important applications of those mechanisms, as in JAKLD/SC in this study. In terms of implementation of garbage collection, Henderson's accurate GC [8], which scans roots in the execution stack as structure fields by using "structure and pointer", employs a similar translation technique. In "structure and pointer" technique, compiler optimizations such as register allocation are inhibited since roots are fields of address-taken structures. One of the goals of L-closures including translators [9], [14], [15], [16] was to mitigate this problem.

The implementations of JAKLD in this paper use nested functions to implement capturing continuations and scanning roots. Each of these nested functions have code for both of these processes and an argument to decide which process should be performed. However, when GC occurs during the process of capturing a continuation, it may fail to scan some roots because GC occurs in the middle of "stack-walking" and then starts root scanning with the nested function at that time (i.e., scanning process interferes with capturing process). To solve this problem, the current implementations remember L-closure (or closure) at the start of capturing a continuation as a global variable, and use it as the starting point of root scanning. Although temporary variables in nested functions may be roots, we took care not to use such variables for the current implementations.

Since capturing continuations and scanning roots are originally different processes, it may be natural to perform these processes by separate nested functions. In this fashion, the problem of interference can be solved by receiving a nested function for scanning roots as an argument to a nested function for capturing continuations. (At the start of capturing, the L-closure or closure for root scanning at that time is passed.) If temporary variables that should be treated as roots are used in nested functions for capturing, an additional nested function that scans these variables should be defined within a nested function for capturing.

As for translating hierarchically defined nested functions, there is an approach to reducing the hierarchy by repeating transformation. To make this work, a translator must be able to retransform transformed programs for the next level of nested functions. A similar approach includes the translation of hierarchical `shift`/`reset` into the CPS hierarchy [5]. Each hierarchical occurrence of `shift`/`reset` has a natural number denoting the level of the hierarchy, and this number is used to avoid interference with the different level of `shift`/`reset`. By repeating CPS transformation for each level of the hierarchy, programs can be translated into ones that contain no `shift`/`reset`. There is also the CPS hierarchy based on operational semantics, which can implement the CPS hierarchy more directly and efficiently [6].

The translations from SC-NF can be considered as *closure conversion* since environments hidden in execution stacks are converted into environments on explicit stacks (or explicit structures). Therefore, if we adopt the repeating transformation approach mentioned above for implementing hierarchical nested functions, the closure conversion is repeated to reduce the hierarchy of nested functions. Another approach using the closure conversion is to hoist hierarchical nested functions to the top-level as a whole after all environments of these nested functions are translated into explicit ones.

In any case, the verification of translation methods for hierarchical nested functions is a future work.

9. Conclusions

In this study, we reimplemented a Scheme implementation JAKLD/XC in an extended S-expression based C language SC-NF featuring nested functions. Based on the new Scheme implementation, we evaluated not only enhanced-C-compiler based implementations of LESA mechanisms but also transformation-based portable implementations.

The Scheme implementation JAKLD/XC was written in an extended C language XC-cube that features language mechanisms for implementing high-level programming languages, such as LESA mechanisms "L-closures" and "closures", with which the running program/process can legitimately access data deeply in execution stacks (C stacks).

L-closures are lightweight lexical closures created from nested function definitions. In addition to enhanced C compilers, we have portable implementations of L-closures, which are translators from an extended S-expression based C language into the standard C language. Closures are standard lexical closures created from nested function definitions. In this study, we newly implemented closures using translators into the standard C language. In this study, by developing a new Scheme implementation JAKLD/SC in an extended SC language SC-NF that features nested functions, we evaluated not only enhanced-C-compiler based implementations of LESA mechanisms but also transformation-based portable implementations, which are translators from an extended SC language SC-NF into the standard C language.

With Scheme programs where the nest of active (pre-return) calls is not so deep and only a limited part of the execution stack is used, transformation-based implementations of L-closures (each based on either the execution stack reconstruction technique [9] or the frame-by-frame restoration technique [14]) achieve 7%~15% lower performance than enhanced C compilers; this performance would be acceptable when considering transformation-based implementations that do not require enhanced C compilers. On the other hand, as in the case of programs written in CPS, with Scheme programs which frequently need to create a space-efficient first-class continuation to support proper tail recursion every time execution stack is about to overflow, transformation-based implementations of L-closures achieve better performance than enhanced C compilers; the frame-by-frame restoration technique is especially effective for reducing the invocation costs as intended.

We also compared transformation-based implementations of L-closures against the enhanced-C-compiler based implementation of closures, which are a mechanism more moderate than L-closures. Closures show remarkably good performance. Neither transformation-based implementations of L-closures (execution stack reconstruction and frame-by-frame restoration) with substantial delay judgment costs can achieve better performance than closures. This result motivated us to develop a new transformation-based implementation of closures. Indeed, the newly developed transformation-based implementation of closures achieved not only portability but also better performance than expected, which was better than enhanced-C-compiler based implementations of closures.

As future work, we would like to study how to reduce the delay judgment costs of transformation-based implementations of L-closures (how to reduce the memory accesses and branch instructions).

Acknowledgments The authors would like to thank Prof. Taiichi Yuasa for his design and implementation of JAKLD, and Seiji Umatani and Hirofumi Kojima who are members of Yuasa laboratory during the development of JAKLD/XC. We would like to especially thank Hidetoshi Kato for his implementation of JAKLD/C. This work was supported in part by JSPS KAKENHI Grant Numbers JP26280023 and JP17K00099.

References

[1] Revised^7 Report on the Algorithmic Language Scheme (2013), available from ⟨http://www.r7rs.org/⟩.
[2] Baker, H.G.: CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A., *ACM SIGPLAN Notices*, Vol.30, pp.17–20 (1995).
[3] Breuel, T.M.: Lexical Closure for C++, *Proc. USENIX C++ Conference*, pp.293–304 (1988).
[4] Clinger, W.D.: Proper Tail Recursion and Space Efficiency, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (*PLDI '98*), pp.174–185 (1998).
[5] Danvy, O. and Filinski, A.: Abstracting Control, *Proc. 1990 ACM Conference on LISP and Functional Programming*, pp.151–160 (online), DOI: 10.1145/91556.91622 (1990).
[6] Danvy, O. and Yang, Z.: An Operational Investigation of the CPS Hierarchy, *Proc. 8th European Symposium on Programming* (*ESOP'99*), Lecture Notes in Computer Science, No.1576, Springer Berlin Heidelberg, pp.224–242 (1999).
[7] Gabriel, R.P.: *Performance and Evaluation of Lisp Systems*, Massachusetts Institute of Technology, Cambridge, MA, USA (1985).
[8] Henderson, F.: Accurate Garbage Collection in an Uncooperative Environment, *Proc. 3rd International Symposium on Memory Management*, pp.150–156 (online), DOI: http://doi.acm.org/10.1145/512429.512449 (2002).
[9] Hiraishi, T., Yasugi, M. and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol.2, pp.262–279 (2006). (*IPSJ Trans. Programming*, Vol.47, No.SIG 6 (PRO 29), pp.50–67 (2006)).
[10] Hiraishi, T., Yasugi, M. and Yuasa, T.: A Reuse Mechanism of Transformation Rules for the SC Language System, *JSSST Computer Software*, Vol.28, No.1, pp.258–271 (2011). (in Japanese).
[11] Kato, H.: A Compact Lisp Language System for Experimenting GC Algorithms, Bachelor's Thesis, Kyoto University (2007). (in Japanese).
[12] Sperber, M., Dybvig, R.K., Flatt, M., Straaten, A.V., Findler, R. and Matthews, J.: Revised^6 Report on the Algorithmic Language Scheme, *Journal of Functional Programming*, Vol.19, pp.1–301 (2009).
[13] Tarditi, D., Lee, P. and Acharya, A.: No Assembly Required: Compiling Standard ML to C, *ACM Letters on Programming Languages and Systems*, Vol.1, pp.161–177 (1992).
[14] Tazuke, M., Yasugi, M., Hiraishi, T. and Umatani, S.: Reducing Invocation Costs of L-Closure, *IPSJ Trans. Programming*, Vol.6, No.2, pp.13–32 (2013). (in Japanese).
[15] Yasugi, M., Hiraishi, T., Shinohara, T. and Yuasa, T.: L-Closure: A Language Mechanism for Implementing Efficient and Safe Programming Languages, *IPSJ Trans. Programming*, Vol.49, No.SIG 1 (PRO 35), pp.63–83 (2008). (in Japanese).
[16] Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proc. 15th International Conference on Compiler Construction* (*CC2006*), Lecture Notes in Computer Science, No.3923, Springer-Verlag, pp.170–184 (2006).
[17] Yasugi, M., Ikeuchi, R., Hiraishi, T., Komiya, T. and Shigemoto, K.: Evaluating Mechanisms for Legitimate Execution Stack Access with a Scheme Interpreter in an Extended SC Language, *The 20th JSSST Workshop on Programming and Programming Languages* (*PPL2018*) (2018). (in Japanese).
[18] Yasugi, M., Kojima, H., Komiya, T., Hiraishi, T., Umatani, S. and Yuasa, T.: A Properly Tail-recursive Scheme Interpreter Using L-Closure, *IPSJ Trans. Programming*, Vol.3, No.5, pp.1–17 (2010). (in Japanese).
[19] Yuasa, T.: A Lisp Driver to Be Embedded in Java Applications, *IPSJ Trans. Programming*, Vol.44, No.SIG 4 (PRO 17), pp.1–16 (2003). (in

Japanese).
[20] Yuasa, T.: Enhancing a Scheme Interpreter JAKLD Written in Java for SICP Exercises, *Proc. 50th Programming Symposium*, pp.51–60 (2009). (in Japanese).

Masahiro Yasugi was born in 1967. He received his B.E. in Electronic Engineering, his M.E. in Electrical Engineering, and his Ph.D. in Information Science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. In 1998–2012, he was an associate professor at Kyoto University. Since 2012, he is a professor at Kyushu Institute of Technology. In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of IPSJ, ACM, and the Japan Society for Software Science and Technology. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award.

Reichi Ikeuchi was born in 1994. He received his Bachelor's degree in Computer Science and Systems Engineering (Artificial Intelligence) from Kyushu Institute of Technology in 2017. Since 2017, he works for UDOM Co., Ltd. His research interests include programming language systems.

Tasuku Hiraishi was born in 1981. He received a B.E. in Information Science in 2003, an M.E. in informatics in 2005, and a Ph.D. in informatics in 2008, all from Kyoto University. In 2007–2008, he was a fellow of the JSPS (at Kyoto University). Since 2008, he has been working at Kyoto University as an assistant professor at Supercomputing Research Laboratory, Academic Center for Computing and Media Studies, Kyoto University. His research interests include parallel programming languages and high performance computing. He won the IPSJ Best Paper Award in 2010. He is a member of IPSJ, JSSST, and ACM.

Tsuneyasu Komiya was born in 1969. He received his B.E., M.E. and Dr.Eng. degrees from Toyohashi University of Technology in 1991, 1993 and 1996, respectively. In 1996–2003, he was a research associate at Kyoto University. In 2003–2006, he was a lecturer at Toyohashi University of Technology. Since 2007, he is an associate professor at the University of Electro-Communications. His research interests include programming languages and their systems. He received an IPSJ best paper award in 1997.