Implementing S-Expression Based Extended Languages in Lisp

Tasuku HIRAISHI
Masahiro YASUGI
Taiichi YUASA

Kyoto University
Introduction

- Many extended C-like languages are implemented by translating them into C (multi-threading, check-pointing, GC, etc.)
  - Much easier than modifying a C compiler.
  - Once implemented, works on various platforms.
Language Extensions by Translation

Extended Language → AST → AST

Programs for Transformation

Variants
Structures
Objects (of OOL)

Convenient for transformation and analysis.

AST = Abstract Syntax Tree
Our Proposal

- Language extensions for *S-expression based C languages* (SC languages).
  - An AST is represented by an S-expression.
  - The S-expression is also used as (part of) a source program.

Convenient for transformation and analysis. **Suitable as a source language.**
Purpose

• Decreasing implementation cost of language extension thanks to:
  – Pre-existing Lisp capabilities for manipulating S-expressions,
  – Easiness of adding new constructs,
  – Natural description of transformation rules,
  – Reusability of (part of) implementation.
Table of Contents

• SC Language System
  – Overview
  – SC-0 Language
  – Transformation Rules

• An Example of a Language Extension
  – Lightweight-SC

• Related Work

• Future Work and Summary
The SC Language System

- The SC Language System
  - Overview
  - The SC-0 Language
  - Transformation Rules
- An Example of a Language Extension
  - Lightweight-SC
- Related Work
- Future Work and Summary
The SC Language System Overview

- A framework for language extensions over SC languages.
- Deals with transformation from extended SCs into C.
- Consists of three modules:
  - The SC compiler
  - The SC translator
  - The SC preprocessor
Further Extended SC

SC preprocessor

SC translator

Transformation Rule-Set B

Extended SC

SC preprocessor

SC translator

Transformation Rule-Set A

SC-0

SC preprocessor

SC compiler

C

SC compiler

SC translator

SC preprocessor

: $\text{SC-0} \rightarrow \text{C}$

: an SC $\rightarrow$ another SC

: preprocess (macro expansion, etc.)
The SC-0 Language

- Semantics of C
- Syntax based on S-expressions.

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

long sum(long *ar, int n){
  long s=0;
  int i=0;
  do{
    if (i >= n) break;
    s += ar[i++];
  } while(1);
  return s;
}
### SC-0 Syntax (for Expressions)

<table>
<thead>
<tr>
<th>C</th>
<th>SC-0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d = a + b * (-c)</code></td>
<td><code> (= d (+ a (* b (- c))))</code></td>
</tr>
<tr>
<td><code>x += 4</code></td>
<td><code> (+= x 4)</code></td>
</tr>
<tr>
<td><code>f (a, b)</code></td>
<td><code> (f a b)</code></td>
</tr>
<tr>
<td><code>(a&gt;b)?a:b</code></td>
<td><code> (if-exp (&gt; a b) a b)</code></td>
</tr>
<tr>
<td><code>b = *pa</code></td>
<td><code> (= b (mref pa))</code></td>
</tr>
<tr>
<td><code>pa = &amp;a</code></td>
<td><code> (= pa (ptr a))</code></td>
</tr>
</tbody>
</table>
### SC-0 Syntax (for Expressions)

<table>
<thead>
<tr>
<th>C</th>
<th>SC-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar[3][4]</td>
<td>(aref ar 3 4)</td>
</tr>
<tr>
<td>st.a</td>
<td>(fref st a)</td>
</tr>
<tr>
<td>sizeof (a)</td>
<td>(sizeof a)</td>
</tr>
<tr>
<td>sizeof (int)</td>
<td>(sizeof int)</td>
</tr>
<tr>
<td>i = (int)d</td>
<td>(= i (cast int d))</td>
</tr>
<tr>
<td>(funarray[3]) (a,b)</td>
<td>((aref funarray 3) a b)</td>
</tr>
</tbody>
</table>
### SC-0 Syntax (for *Statements*)

<table>
<thead>
<tr>
<th>C</th>
<th>SC-0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (a&gt;0)</code></td>
<td><code>(if (&gt; a 0)</code></td>
</tr>
<tr>
<td></td>
<td><code>(inc a)</code></td>
</tr>
<tr>
<td></td>
<td><code>(dec a)</code> )</td>
</tr>
<tr>
<td><code>a++;</code></td>
<td></td>
</tr>
<tr>
<td><code>else a--;</code></td>
<td></td>
</tr>
<tr>
<td><code>switch (n) {</code></td>
<td><code>(switch n</code></td>
</tr>
<tr>
<td><code>  case 1: ... break;</code></td>
<td><code>(case 1) ... (break)</code></td>
</tr>
<tr>
<td><code>  case 2: ... break;</code></td>
<td><code>(case 2) ... (break)</code></td>
</tr>
<tr>
<td><code>  default: ...</code></td>
<td>`(default) ... )</td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>
### SC-0 Syntax (for Declarations)

<table>
<thead>
<tr>
<th>C</th>
<th>SC-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a=10;</td>
<td>(def a int 10)</td>
</tr>
<tr>
<td>static *ps;</td>
<td>(static ps (ptr int))</td>
</tr>
<tr>
<td>int sqr (long x)</td>
<td>(def (sqr x) (fn int long)</td>
</tr>
<tr>
<td>{ return x*x; }</td>
<td>(return (* x x)) )</td>
</tr>
<tr>
<td>void foo (int x){}</td>
<td>(def (foo x) (fn void int))</td>
</tr>
<tr>
<td>void foo (int);</td>
<td>(decl foo (fn void int))</td>
</tr>
</tbody>
</table>
### SC-0 Syntax (for Declarations)

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>SC-0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>struct strab {</td>
<td>(def (struct strab)</td>
</tr>
<tr>
<td></td>
<td>int a;</td>
<td>(def a int)</td>
</tr>
<tr>
<td></td>
<td>long b;</td>
<td>(def b long)</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td></td>
</tr>
<tr>
<td></td>
<td>typedef int *int_p;</td>
<td>(deftype int-p (ptr int))</td>
</tr>
<tr>
<td></td>
<td>typedef char str[256];</td>
<td>(deftype str (array int 256))</td>
</tr>
</tbody>
</table>
SC-0 Syntax (for *Type-Expressions*)

Type description is more readable.

<table>
<thead>
<tr>
<th>C</th>
<th>SC-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef void</td>
<td>(deftype gg-t</td>
</tr>
<tr>
<td>**(**gg_t)</td>
<td>(ptr (fn (ptr void) long long))</td>
</tr>
<tr>
<td>(void **)(int,int)))(long,long);</td>
<td>(ptr (fn (ptr void) int int)))</td>
</tr>
</tbody>
</table>
The SC Preprocessor

- Corresponds to the C preprocessor.
  - (%include file-name)
  - (%defmacro name lambda-list . body)
  - (%defconstant name Sexpr)
  - (%ifndef symbol body1 body2)
  - (%ifndef symbol body1 body2)
  - (%ifdef symbol body1 body2)
  - (%ifndef symbol body1 body2)
  - (%cinclude C-header-file-name)
    - for using printf, NULL, etc.
The SC Translator

- Interprets transformation rules for transforming S-expressions.
- The input/output S-expression is:
  - An extended SC program,
  - An SC-0 program, or
  - An intermediate data structure.
Transformation Rules

- Defined as pattern-matching functions over their arguments.
- The SC translator compiles rules into usual Common Lisp function definitions.
Writing Transformation Rules

(STAT (begin ,@rem))
  -> `((begin ,@(BODY rem)) )
(STAT (if ,exp ,@rem))
  -> `((if ,(EXPR exp)
            ,@(mapcar #'(lambda (st) (car (STAT st))) rem)) )
(STAT (switch ,exp ,@rem) )
  -> `((switch ,(EXPR exp) ,@(BODY rem)) )
(STAT (while ,exp ,@rem) )
  -> (let ((cdt (EXPR exp)))
        `((if ,cdt
            (do-while ,cdt ,@(BODY rem)))) )
(STAT (loop ,@rem) )
  -> `((do-while 1 ,@(BODY rem)) )
...
Applying Transformation Rules

(F (loop, @body))
-> (do-while 1, @body)
(F (while, cond, @body))
-> (if, cond (do-while, cond, @body))

(F (while (< i 10) (++ i) (-- j))) = ?
Pattern: (while, @body, @body)
Argument: (while (< i 10) (++ i) (-- j))
Applying Transformation Rules

\[(F \ (\text{loop} \ , \ @\text{body}))\]
\[\rightarrow \ '(\text{do-while} \ 1 \ , \ @\text{body})\]
\[(F \ (\text{while} \ , \ \text{cond} \ , \ @\text{body}))\]
\[\rightarrow \ '(\text{if} \ , \ \text{cond} \ (\text{do-while} \ , \ \text{cond} \ , \ @\text{body}))\]

\[\text{cond} \leftarrow \ (< \ i \ 10)\]
\[\text{body} \leftarrow \ ((++ \ i) \ (-- \ j))\]

\[(F \ (\text{while} \ (< \ i \ 10) \ (++ \ i) \ (-- \ j)))\]
\[= \ (\text{if} \ , \ \text{cond} \ 10)\]
\[\ (\text{do-while} \ (\text{cond10}) \ (@\text{body} \ (-- \ j)))\]
An Example of a Language Extension

✓ The SC Language System
  – Overview
  – The SC-0 Language
  – Transformation Rules

➢ An Example of a Language Extension
  – Lightweight-SC

• Related Work
• Future Work and Summary
LW-SC (Lightweight-SC)

- SC-0 + nested functions

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

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

- Translate LW-SC into SC-0.

How nested functions access local variables of their owner?

(\texttt{def (foo) (fn ...)}
 \texttt{(def a int)}
 \texttt{(def (g1) (lightweight ...)}
 \texttt{... (return a))})

\textbf{Wrong:}

(\texttt{def (foo) (fn ...)}
 \texttt{(def a int)}
 \texttt{...)}
 \texttt{(def (g1-in-foo) (fn ...)}
 \texttt{... (return a))})

\textbf{Correct:}

(\texttt{def (foo) (fn ...)}
 \texttt{(def a int)}
 \texttt{... (return a))})
Naïve Implementation

- Each generated C program employs an explicit stack.
- The explicit stack saves local variables, arguments, etc.
- Access owner’s local variable can be accessed through a frame pointer on the explicit stack, which is passed as an additional parameter.

```c
foo(...) {
    int a;
    INITIALIZE_FRAME_PTR
    ... ... }
g1_in_foo (struct foo_frame *pfp, ...) {
    ... ... return pfp->a; }
```
Implementation with *Lightweight Closures*

- Explicit stack is referred to only when nested functions are actually invoked.
- When “nested function” calls occur, the explicit stack is validated (by temporarily returning executing functions).
Implementation with *Lightweight* Closures

- Explicit stack is referred to only when nested functions are actually invoked.
- When “nested function” calls occur, the explicit stack is validated (by temporarily returning executing functions).

```
foo(...)
{
    int a;
    INITIALIZE_FRAME_PTR
    ... ...
}
g1_in_foo
(struct foo_frame *pfp, ...)
{
    ... ...
    return pfp->a;
}
```
Implementation with *Lightweight Closures*

- When returning from the nested function, reconstruct the execution stack restoring the local variables, the parameters, and the execution points.
Translation from LW-SC to SC-0

Translation divided into four phases (rule-sets):

1. **type** rule-set: Adds type information to all expressions.
2. **temp** rule-set: Transforms in such a way that no function call appears as a subexpression.
3. **lightweight** rule-set: The main transformation
4. **untype** rule-set: Removes the type information added by type rule-set.
Phase 1: Type Rule-Set

- Transforms each *expression* into
  
  \[(\text{the type-expression expression})\]

- Adds the symbol "call" at the head of each function call.

\[(\text{def (h x) (fn double double)} \\leftarrow \text{def y int 10} \\leftarrow \text{return (+ y (f x))})\]

\[(\text{def (h x) (fn double double)} \\leftarrow \text{def y int 10} \\leftarrow \text{return (the double (+ (the int y) (the double (call (the (fn double double) f) (the double x))))))}\]
Phase 2: Temp Rule-Set

- \((f (g x)) \rightarrow (= \text{tmp} (g x))\)
  
  \((f \text{tmp})\)
- Adds declarations for the *temporary* variables.

\[
\begin{align*}
\text{(def (h x) (fn double double))} \\
\text{(def y int 10)} \\
\text{(return (the double (+ (the int y)) (the double (call (the (fn double double) f) (the double x))))))}
\end{align*}
\]

\[
\begin{align*}
\text{(def (h x) (fn double double))} \\
\text{(def y int 10)} \\
\text{(def tmp double)} \\
\text{(the double (= (the double tmp) (the double (call (the (fn double double) f) (the double x))))))} \\
\text{(return (+ (the int y) (the double tmp))))))}
\end{align*}
\]
Phase 3: Lightweight Rule-Set

- Moves all definitions of nested functions to be top-level definitions.
- Adds definitions of special variables/functions.
- The other transformation needed for:
  - invocation of ordinary/nested functions,
  - returning from functions,
  - function definitions.
Phase 4: Untype Rule-Set

- Removes type information to generate correct SC-0 code.

```
(def (h x) (fn double double)
 (def y int 10)
 (def tmp double)
 (the double
  (= (the double tmp)
   (the double
    (call (the (fn double double) f)
     (the double x))))))
 (return (+ (the int y)
            (the double tmp))))
```

```
(def (h x) (fn double double)
 (def y int 10)
 (def tmp double)
 (= tmp (f x))
 (return (+ y tmp)))
```
Performance

• The GNU C Compiler also provides nested functions as an extension to C (implemented as an extended C compiler).

• Compare allocation/maintenance overhead
  – UltraSPARC-III (1.05GHz) and Pentium 4 (3GHz)
  – GCC with -O2 optimizers as a backend for SC.
Performance

Time Relative to plain C

UltraSPARC-III  Pentium 4

UltraSPARC-III  Pentium 4
Implementation Cost

• The number of lines of each rule-set:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>450</td>
</tr>
<tr>
<td>temp</td>
<td>340</td>
</tr>
<tr>
<td>lightweight</td>
<td>780</td>
</tr>
<tr>
<td>untype</td>
<td>10</td>
</tr>
</tbody>
</table>

• The rule-sets type, temp and untype are reusable for many other extensions.

• Generated C code can be compiled by most C compilers.
Application of LW-SC

- Multi-threading
- Check-pointing
- Copying GC
- Load balancing
Implementation of Copying GC

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

(def (randsearch scan0 this n) (fn void sht (ptr Bintree) int)
  (def (scan1) (lightweight void void) ; nested function
    (= this (move this))
    (scan0))

(decl i int)
(decl k int)
(decl seed (array unsigned-short 3))
(= (aref seed 0) 8) (= (aref seed 1) 9)
(= (aref seed 2) 10)
(for ((= i 0) (< i n) (inc i))
  (= k (nrand48 seed))
  (search scan1 this k 0))) ; pass scan1 as an additional arg
Related Work

- The SC Language System
  - Overview
  - The SC-0 Language
  - Transformation Rules
- An Example of a Language Extension
  - Lightweight-SC

- Related Work
  - Future Work and Summary
Related Work

- Cilk, OpenMP, etc.
  - Extended Language $\rightarrow$ AST $\rightarrow$ ... $\rightarrow$ AST $\rightarrow$ C
  - Not a framework for general language extensions.
Related Work

- Reflection, compile-time reflection
  - kinds of language extensions.
  - manipulating behaviors of a running program by referring to or modifying meta-level information.
  - Compile-time reflection is similar to our approach,
  - but we provide a more generic framework to transform program.
Related Work

• Pre-Scheme
  – a dialect of Scheme
  – allows low-level machine access of C
    (lacks some features of Scheme such as GC, full
     proper tail recursion, etc.)
  – SC is much closer to C.
Future Work and Summary

✓ The SC Language System
  - Overview
  - The SC-0 Language
  - Transformation rules
✓ An Example of a Language Extension
  - Lightweight-SC
✓ Related Work
  - Future Work and Summary
Future Work

• Debugging support for extended SC programmers.
  – solved by making transformation rules weave debugging code into their output.

• Integrating (independently developed) two or more extensions.

• Providing advanced services based on LW-SC:
  – Copying GC,
  – Check-pointing,
  – Load balancing.
Summary

- A scheme for extending the C language using S-expression based C languages.
- An example of a language extension – LW-SC

- Highly flexible language extensions can be achieved at low implementation cost.