Implementing S-Expression Based Extended Languages in Lisp

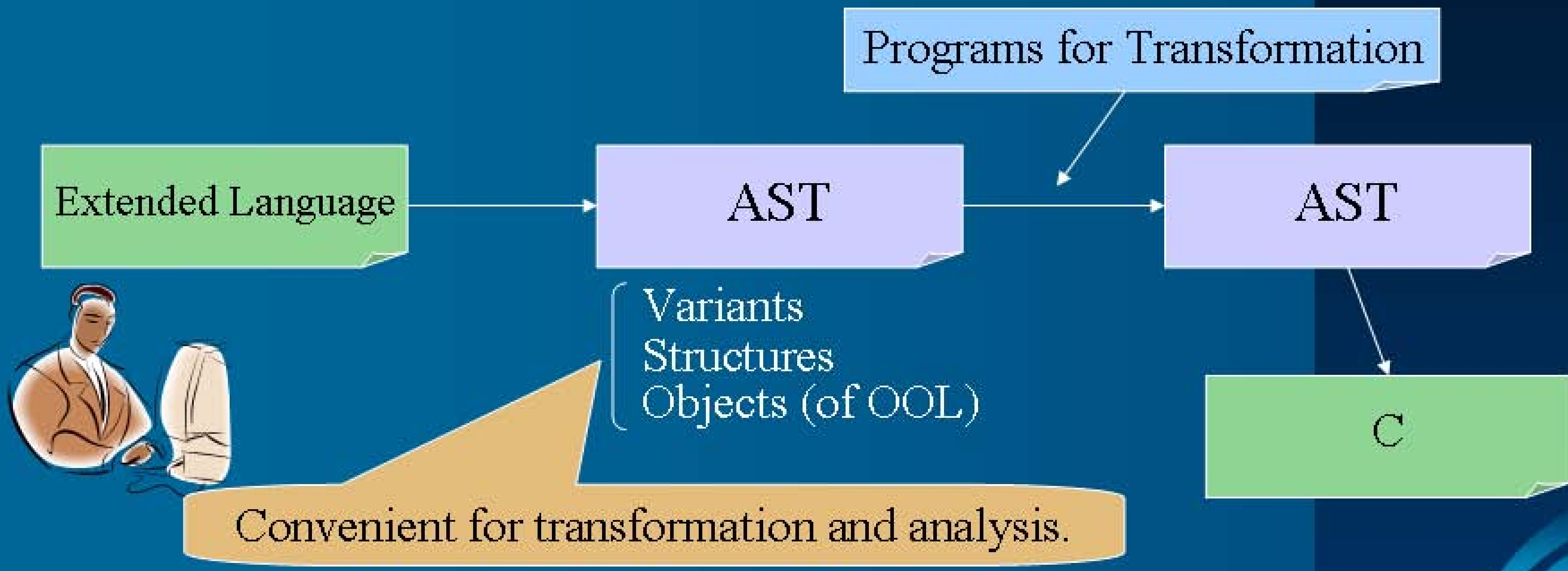
Tasuku HIRAISHI
Masahiro YASUGI
Taiichi YUASA

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.

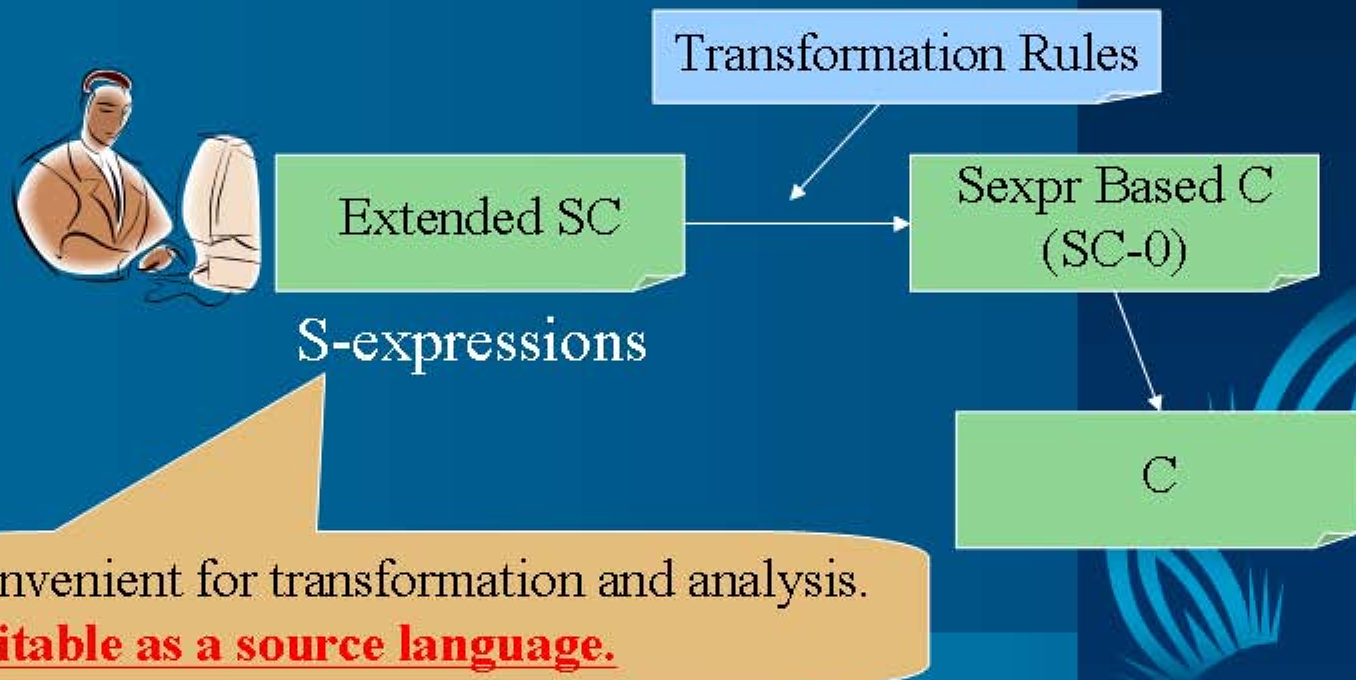
Language Extensions by Translation



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.



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

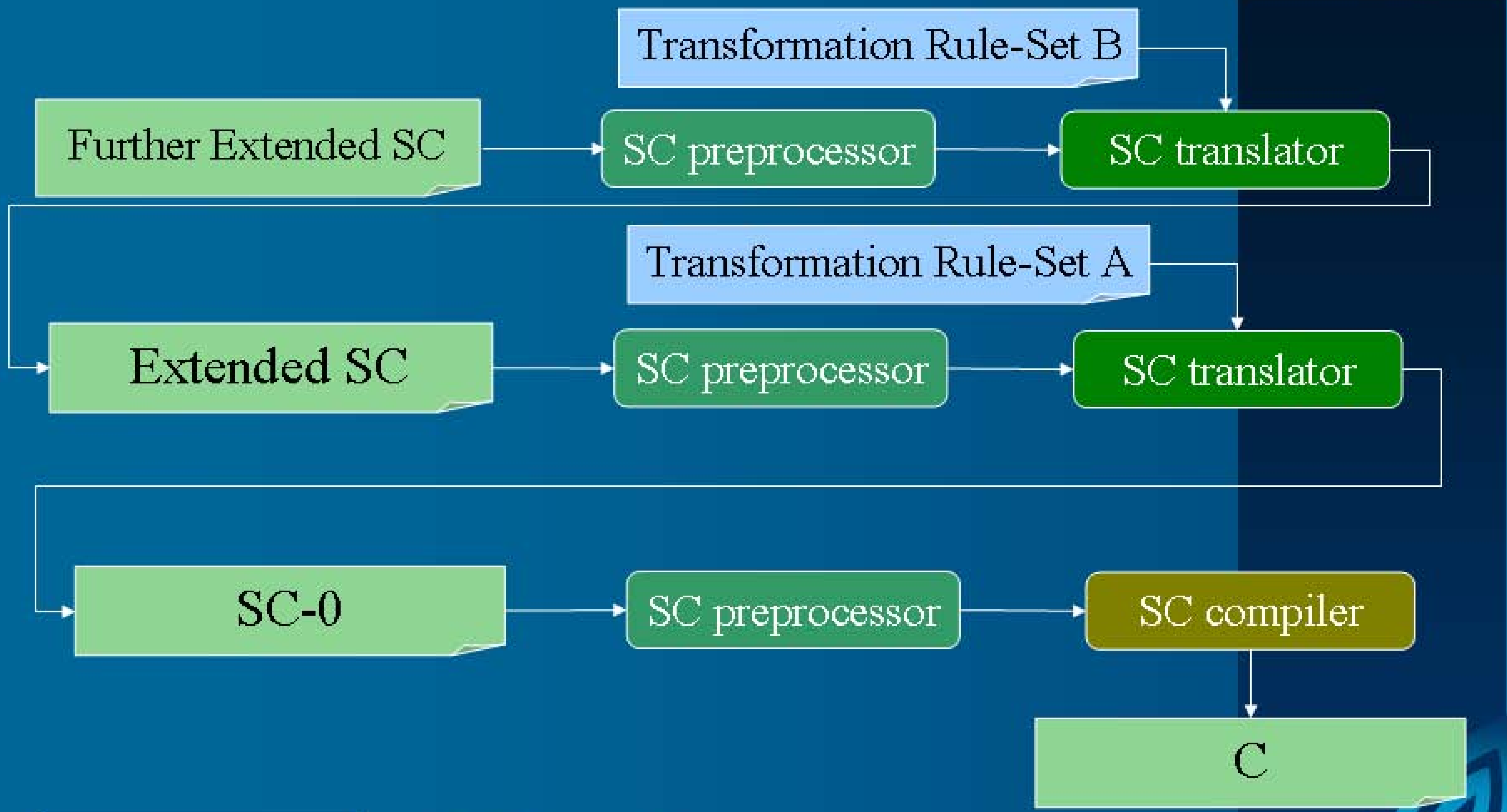
- SC Language System
  - Overview
  - SC-0 Language
  - Transformation Rules
- An Example of a Language Extension
  - Lightweight-SC
- Related Work
- Future Work and Summary

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



- SC compiler : SC-0 → C
- SC translator : an SC → another SC
- SC preprocessor : 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*)

C	*SC-0*
d = a + b * (-c)	(= d (+ a (* b (- c))))
x += 4	(+= x 4)
f (a, b)	(f a b)
(a>b)?a:b	(if-exp (> a b) a b)
b = *pa	(= b (mref pa))
pa = &a	(= pa (ptr a))

SC-0 Syntax (for *Expressions*)

C	*SC-0*
ar[3][4]	(aref ar 3 4)
st.a	(fref st a)
sizeof (a)	(sizeof a)
sizeof (int)	(sizeof int)
i = (int)d	(= i (cast int d))
(funarray[3]) (a,b)	((aref funarray 3) a b)

SC-0 Syntax (for *Statements*)

C	*SC-0*

if (a>0)
  a++;
else a--;
	(if (> a 0)
	    (inc a)
	    (dec a))

switch (n) {
  case 1: ... break;
  case 2: ... break;
  default: ...
}
	(switch n
	  (case 1) ... (break)
	  (case 2) ... (break)
	  (default) ...)

SC-0 Syntax (for *Declarations*)

C	*SC-0*
int a=10;	(def a int 10)
static *ps;	(static ps (ptr int))
int sqr (long x) { return x*x; }	(def (sqr x) (fn int long) (return (* x x)))
void foo (int x){}	(def (foo x) (fn void int))
void foo (int);	(decl foo (fn void int))

SC-0 Syntax (for *Declarations*)

C	*SC-0*

struct strab {
  int a;
  long b;
};
	(def (struct strab)
	  (def a int)
	  (def b long))

typedef int *int_p;	(deftype int-p (ptr int))
typedef char str[256];	(deftype str (array int 256))

SC-0 Syntax (for *Type-Expressions*)

Type description is more readable.

C	*SC-0*

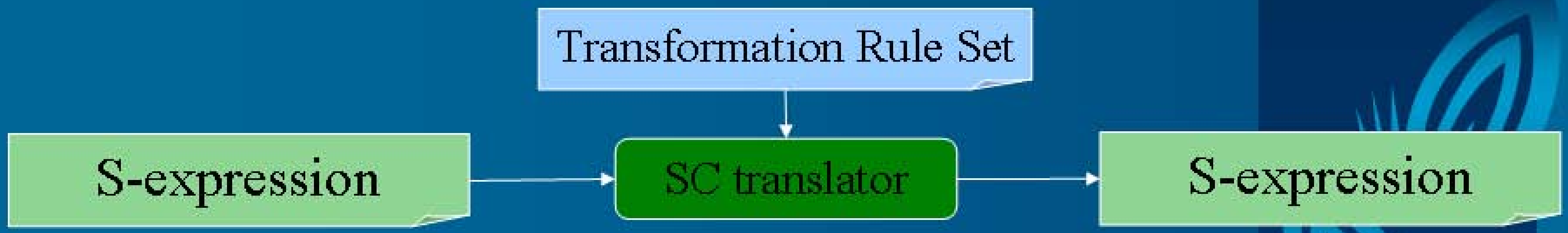
typedef void
`*(*(*gg_t)`
`(void *(*)(int,int)))(long,long);`
	(deftype gg-t
	(ptr (fn
	(ptr (fn (ptr void) long long))
	(ptr (fn (ptr void) int int))))

The SC Preprocessor

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

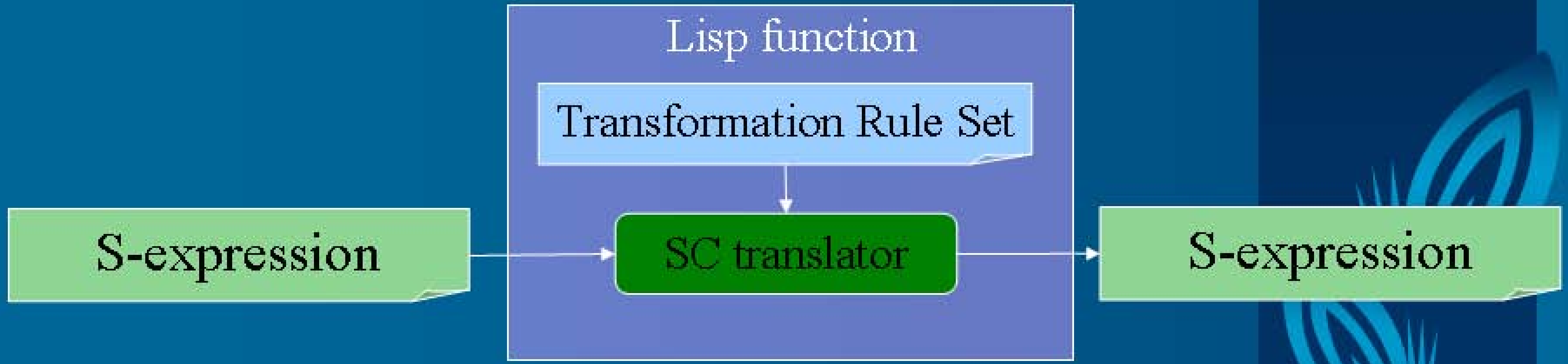
The SC Translator

- 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

Backquote-macro-like notations for *patterns*.

```
(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 ,cond ,@body )` / `(loop ,@body)`

Argument: `(while (< i 10) (++ i) (-- j))`

Applying Transformation Rules

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

```
cond ← (< i 10)

body ← ((++ i) (-- j))
```

```
(F  (while (< i 10) (++ i) (-- j)))
  =  ((iff ,(comd 10)
       (do-while (comd10) ((@bodv (-- 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

```
(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)) )
```

nested function

Implementing Nested Functions

- Translate LW-SC into SC-0.

How nested functions access local variables of their owner?

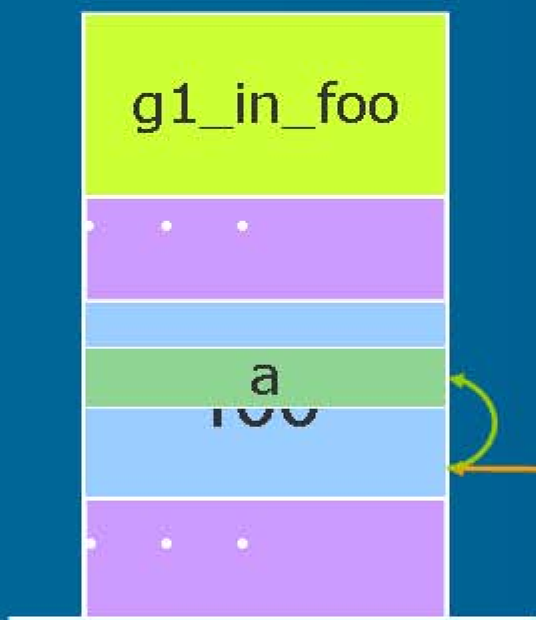
```
(def (foo) (fn ...)
  (def a int)
  (def (g1) (lightweight ...)
    ...
    (return a))
  ...)
```

```
(def (foo) (fn ...)
  (def a int)
  ...)

(def (g1-in-foo) (fn ...)
  ...
  (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.

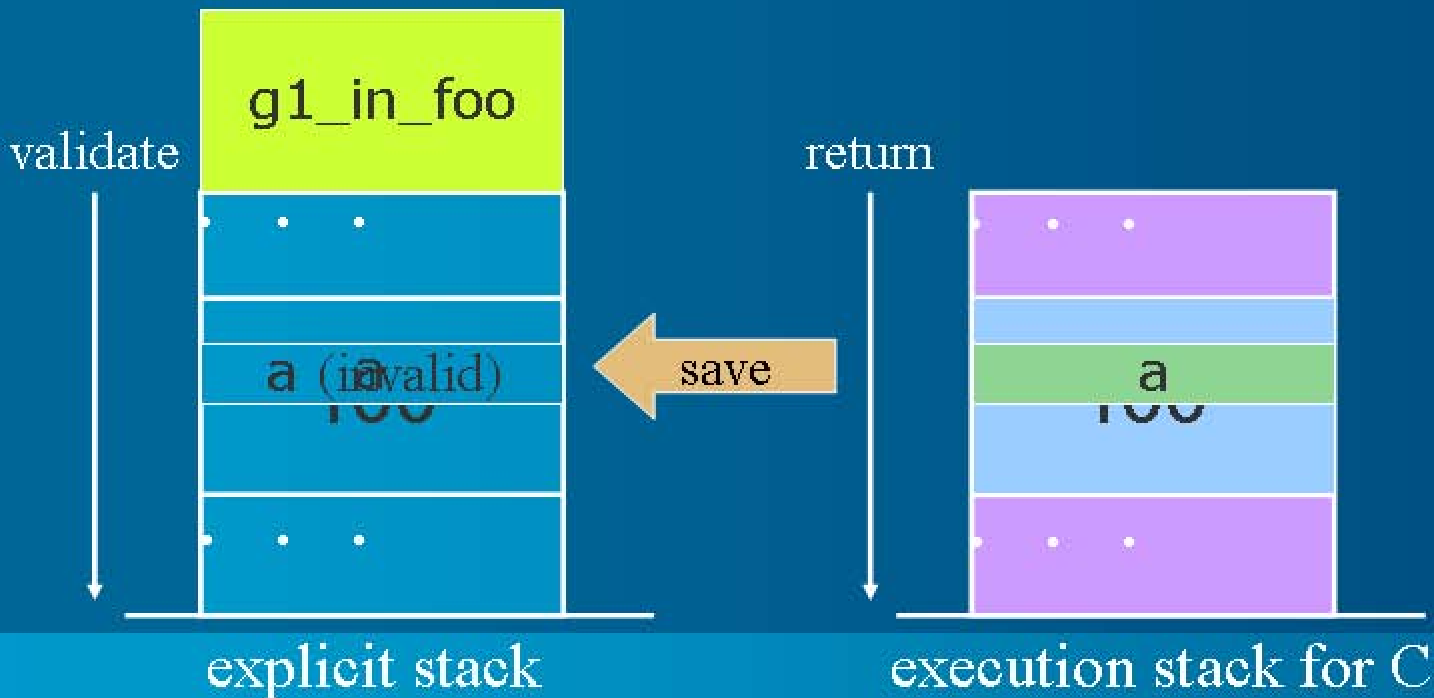


explicit stack

```
foo(…)
{ int a;
  INITIALIZE_FRAME_PTR
  … … }
g1_in_foo
 (struct foo_frame *pfp, …)
{  … …
   return pfp->a;  }
```

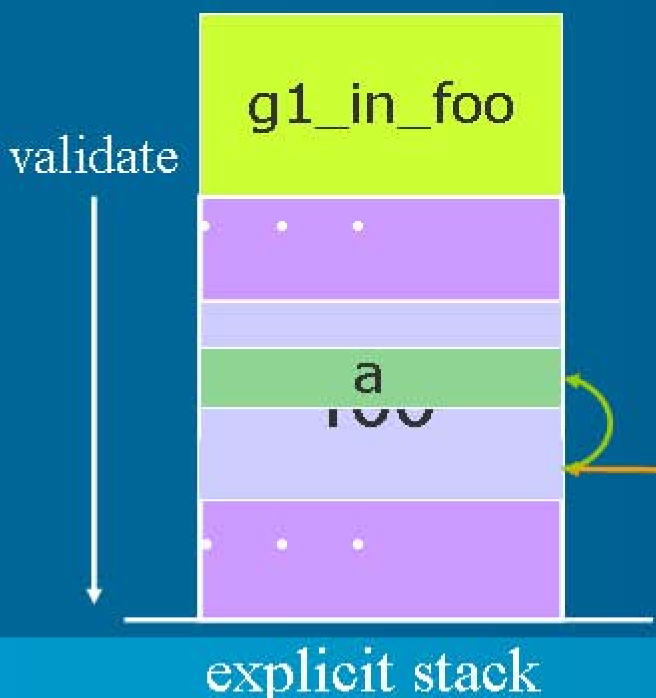
Implementation with *Lightweight* Closures

- 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

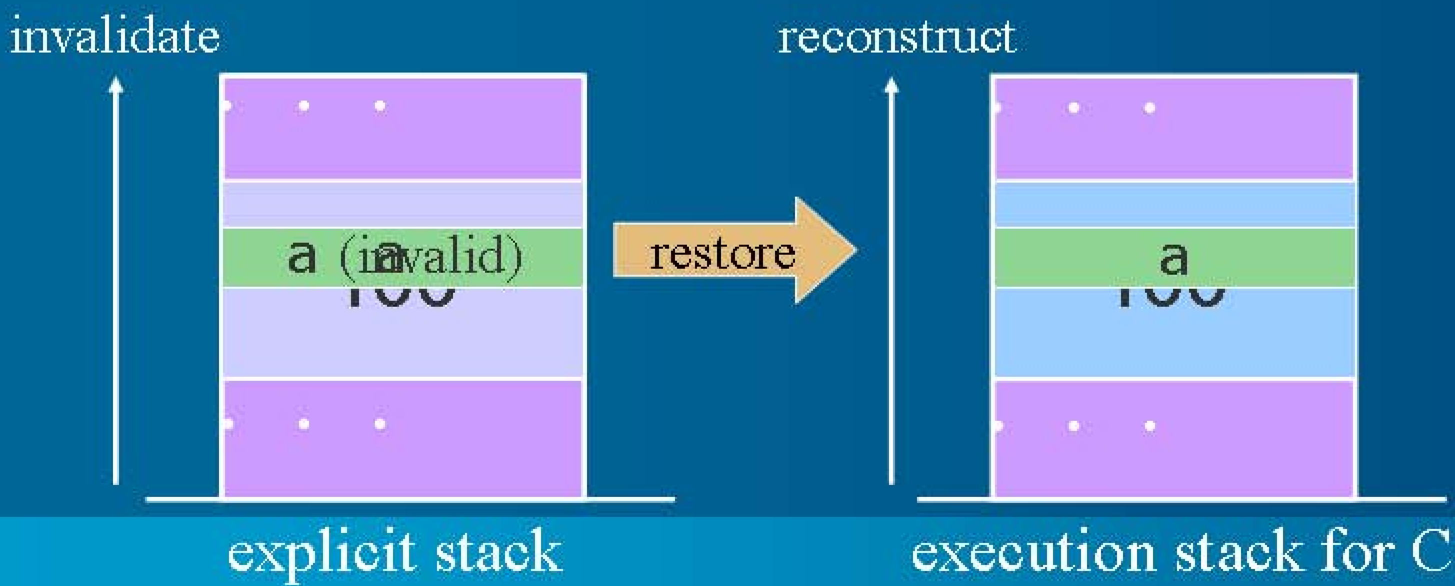
- Explicit stack is referred to only when nested functions are actually invoked.
- When “nested function” calls occur, the explicit stack is validated (by temporarily returning executing functions).




```
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

  (the *type-expression expression*).
- Adds the symbol "call" at the head of each function call.

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

→

```
(def (h x) (fn double double)
  (def y int 10)
  (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)) → (= tmp (g x))
  (f tmp)
- Adds declarations for the *temporary* variables.

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

→

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

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)))))
```

→

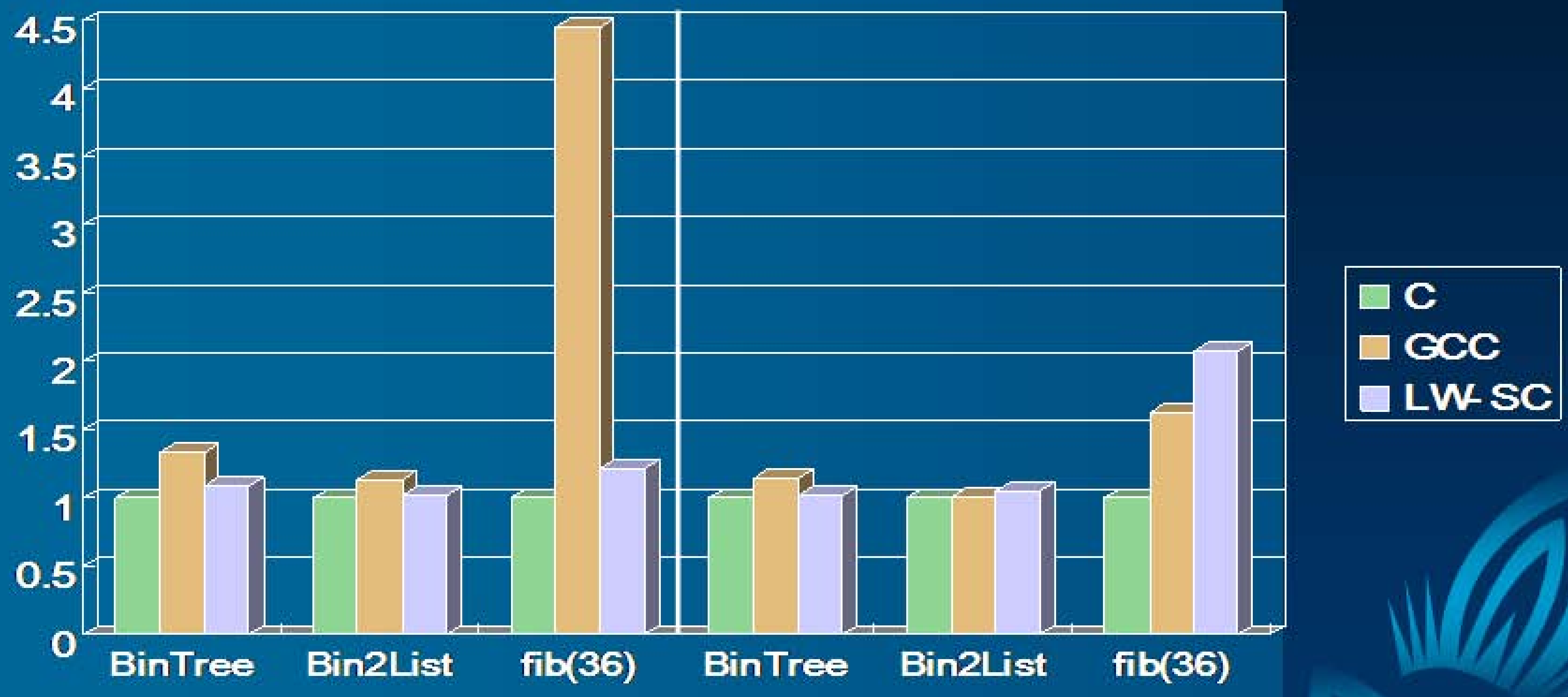
```
(def (h x) (fn double double)
  (def y int 10)
  (def tmp double)
  (= tmp (f x))
  (return (+ y tmp)))
```

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

Implementation Cost

- The number of lines of each rule-set:

type	450
temp	340
lightweight	780
untype	10

- 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))                ; root scan
    (scan0))                            ; scan for caller
  (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 adidtional 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 → AST → ... → AST → 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.