Transformation-based Implementation of S-expression Based C Languages

by
Tasuku Hiraishi

Abstract

This thesis presents schemes for transformation-based implementation of language extension to the C language, which reduce implementation costs and enhance the capability of language extension by translation into C. We also present its practical applications.

The C language is often indispensable for developing practical systems, but it is not an easy task to extend the C language by adding a new feature such as fine-grained multi-threading. We can implement language extension by modifying a C compiler, but sometimes we can do it by translating an extended C program into C code. In the latter method, we usually convert the source program to an Abstract Syntax Tree (AST), apply analysis or transformation necessary for the extension, and then generate C code. Structures, objects (in object-oriented languages), or variants are traditionally used as the data structure for an AST. In this research, we propose a new scheme where an AST is represented by an S-expression and such an S-expression is also used as (part of) a program. For this purpose we have designed the *SC* language, the C language with an S-expression based syntax. This scheme allows rapid prototyping of language extension because (1) adding new constructs to the SC language is easy, (2) S-expressions can easily be read/printed, analyzed, and transformed in the Common Lisp language, which features dynamic variables useful for translation. Since pattern-matching cannot be described directly in Common Lisp, we also propose describing transformation rules with patterns using the backquote-macro notation. Furthermore, we provide features to extend an existing transformation phase (rule-set). This enables us to implement many transformation rule-sets only by describing the difference, and helps us use commonly-used rule-sets as part of the entire transformation.

This thesis also presents the *LW-SC* (LightWeight-SC) language as an important application of this system, which features nested functions (i.e., a function defined inside another function). Without returning from a function, the function can manipulate its caller's local variables (or local variables of its indirect callers) by indirectly calling a nested function of its (indirect) caller. Thus, many high-level services that require "stack

walk" can be implemented by using LW-SC as an intermediate language, with the maintained advantages of transformation-based implementation such as portability and lower implementation cost. Moreover, such services can be efficiently implemented because we design and implement LW-SC so that it provides "lightweight" nested functions by aggressively reducing the costs of creating and maintaining nested functions.

Actually we implemented several features such as multi-threading and copying garbage collection by extending LW-SC. The *Tascell* language is significant one of such applications. Another significant application is Tascell, which realizes our new dynamic load balancing scheme—backtracking-based load balancing. A worker basically performs a computation sequentially, but when it receives a task request from another idle worker, it creates a new task by dividing the remaining computation and returns the new task. In order to reduce the total number of created tasks, each task should be as large as possible. In our scheme, the worker achieves this by performing temporary backtracking before creating the task. What make our scheme better than LTC-based implementations of multi-threaded languages such as MultiLisp and Cilk are elimination of unnecessary concurrency and lazy allocation of working spaces; our approach does not create multiple logical threads as potential tasks or does not need to manage a queue for them. Therefore the overhead can be quite low. In a sequential computation, a single working space can be reused naturally and spatial locality can be improved. Furthermore, programs for various search problems can be written elegantly and run very efficiently even in fine-grained computations because they can delay copying between working spaces by using backtracking. We prototyped a programming language and a framework that realize these ideas. Programmers can write a worker program based on an existing sequential program augmented with new constructs in this language. Our approach also enables a single program to run in both shared and distributed (and also hybrid) memory environments with reasonable efficiency.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my greatest supervisor, Professor Taiichi Yuasa, for his invaluable supports and directing me to this interesting research area.

I would also like to my Prof. Masahiro Yasugi, who has led my research. He is great resources for discussing ideas and concerns. He is extremely helpful and willing to dedicate time to helping me.

I also deeply appreciate reviewers of my thesis, Prof. Masahiko Sato and Prof. Hiroshi Okuno, who gave me valuable advice.

The SC project has been a team effort and I am indebted to all the people who have contributed in some way to the SC language system.

Finally, I would like to thank my family. Their support has always been important to me.

The research was supported in part by the 21st century COE program in Japan.

Chapter 1

Introduction

1.1 Background

The C language is indispensable for developing practical systems, and it is often the case that a new language with features such as fine-grained multi-threading is developed as an extended C language. We can implement such a language extension by modifying a C compiler, but sometimes we can do it by translating an extended C program to an Abstract Syntax Tree (AST), applying analysis or transformation necessary for the extension, and then generating C code. Structures, objects (in object-oriented languages), or variants are traditionally used as the data structure for an AST. Actually, Cilk [10] and OPA [49] are implemented in this way.

Such a transformation-based strategy reduces implementation costs of language extension because implementing a translator is much easier than modifying a C compiler and we need not to implement for various platforms.

However, there remains a problem from the viewpoint of implementation cost; we need to implement not only programs for transformation but also a lexer and a parser for each extension. This problem is significant especially when adding new constructs for the extended language, because we must modify the lexer and the parser each time we change the specification.

The Lisp language [42] allows us to add new constructs to itself easily by its powerful macro facility and its S-expression based syntax; an S-expression, a syntactic unit in

Lisp, is not only suitable for analysis and transformation but also easy to hand-code in. You can implement even a totally new language as a translator to Lisp using the macro facility [12]. Unfortunately, since Lisp is a highly abstract language, it is not applicable to developing language features which require low level operations (such as address operations).

Note that even C has such a problem. Though C is much less abstract than Lisp, it lacks some ability necessary for implementing some kind of language features. For example, when implementing high-level services with "stack walk" such as capturing a stack state for check-pointing and scanning roots for copying GC (Garbage Collection), we need to access variables "sleeping" in the execution stack (variables located below a current frame). But a standard C program cannot do it without taking "memory" addresses of the variables. Traditionally, this problem is handled by using assembly operations as part of generated code at the cost of portability, by managing execution stack in the heap at the cost of efficiency, or by service-specific and elaborate implementation techniques.

Another solution to this problem is to use a language other than C as a target of the translation. For instance, the C-- language [23, 35] is developed for this purpose and has the ability to access sleeping variables by using special operations for "stack walk." We also developed XC-cube, an extended C language with some primitives added for safe and efficient shared memory programming [58]. XC-cube programs can access sleeping variables by using *nested functions* [59], functions defined inside another function. These languages are portable and work efficiently, but are less commonly-used than C—we would like standard C to adopt some features of XC-cube, but that seems hard in the immediate future.

1.2 Our Proposal

This thesis proposes an extension scheme for *SC* languages (extended/plain C languages with an S-expression based syntax), in which each AST is represented by an S-expression and such S-expressions are also used as (part of) a program. This scheme enables us to implement various language extensions to C at low cost because it is easy to add new

constructs to an SC language, and because we can make use of the existing Common Lisp capabilities for reading/printing, analyzing, and transforming S-expressions. We also developed the SC language system in Common Lisp. In this language system, extensions are implemented by transformation rules over S-expressions, that is, Lisp functions which perform pattern-matching on S-expressions.

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

We also propose a solution to the above-mentioned problem which arises from C's inability to access variables sleeping in the execution stack. Our solution uses nested functions which are translated into standard C functions. Each function can manipulate its caller's local variables (or local variables of its indirect callers) sleeping in the execution stack by calling a nested function of its caller. For this purpose, we implemented the *LW-SC* (Lightweight SC) language, which features nested functions as a language extension to C. Many high-level services with "stack walk" can be easily and elegantly implemented by using LW-SC as an intermediate language. Moreover, such services can be efficiently implemented because we design and implement LW-SC to provide "lightweight" nested functions by aggressively reducing the costs of creating and maintaining nested functions. We can also preserve portability because LW-SC is implemented as a translator to standard C,

We presents some practical examples of language extensions which actually use LW-SC, which include a significant example which realizes our novel idea on dynamic load balancing.

1.3 Contributions

This thesis makes the following contributions:

- The *SC-1* language, the language with the C semantics and an S-expression based syntax, gives programmers the option to write C program in the alternative syntax; in particular, seasoned Lisp programmer will prefer the syntax. SC-1 takes advantage of the S-expression based notation. For example, declarations/definitions with

type expressions in SC-1 are more readable than those in C, and SC programmers can use a powerful macro facility.

- The SC-1 language also enables language developers to rapid-prototype their language in cases in which the language can be implemented by translation into C. The implementation can be done by transformation over S-expressions, from the being developed language to SC-1, which is easily done by using the Lisp capabilities. Furthermore, the SC language system provides the following extensions to Common Lisp which help implementation of transformation:

  - The pattern-matching facility over S-expressions. We adopted intuitive backquote-macro-like notation in consideration of symmetry between patterns and expressions.
  - The facility to implement a transformation phase by extending existing phases, like classes of object-oriented languages. This enables us to implement transformation phases only by describing the difference.

  Some of these extensions are useful for general Lisp programming.

- The LW-SC language, an extended SC-1 language which features nested function, enables us to implement high-level services which require "stack walk," in a unified way. The implementations are portable and efficient.

- *C2SC Compiler*, a translator from C to SC-1, enables SC programmers to use existing C header files. It can translate type information declared in C header files and, unlike existing foreign function interfaces, enables us to use some `#define` macros by translating them into SC macros. Our ideas here can also be applied for other languages.

- *Backtracking-based load balancing*, our new dynamic load balancing scheme, is realized by the *Tascell* language, which is implemented using the SC language system and LW-SC. This scheme achieves quite low overhead for load balancing and improves spatial locality in a sequential computation. Furthermore, programs for

various search problems can be written elegantly and run very efficiently even in fine-grained computations because they can delay copying between working spaces by using backtracking. Our approach also enables a single program to run in both shared and distributed (and also hybrid) memory environments with reasonable efficiency. We evaluated the performance and showed that Tascell runs more efficiently than Cilk in many applications.

1.4 Organization of the Thesis

The organization of this thesis is as follows.

Chapter 2 describes the SC language system. We first give an overview, and then show the details of its components and the base SC languages of the system. We also show how to implement language extensions using this system.

Chapter 3 evaluates our scheme for implementing language extension from the viewpoint of ease of use and implementation cost.

Chapter 4 introduces the LW-SC language as a practical implementation example of language extension using the SC language system. We also show implementation techniques for "lightweight" nested functions. As mentioned above, this extension is not just an example but has its own significance; it enables us to implement many high-level services easily and elegantly by using LW-SC as an intermediate language. In fact, multi-threading and copying GC (Garbage Collection) are described as examples of such implementations.

Chapter 5 introduces *C2SC Compiler*, a C-to-SC translator. Such a "reverse" translator is needed when SC requires header files corresponding to existing C header files. Despite the equivalence of the semantics of C and SC-1, the translation is not obvious. In particular, it is not always possible to translate `#define` macro definitions mainly because C allows syntactically incomplete token strings as an expanded code. This chapter discusses the limitations of the translation and proposes pragmatic and reasonable solutions to them.

Chapter 6 describes an attractive application of the SC language system and the LW-SC language, the *Tascell* language. Tascell is a parallel language which realizes

our novel idea on dynamic load balancing, backtracking-based load balancing. Thus Chapter 6 is written for more a proposition of the new scheme than an illustration of an implementation of language extension.

Chapter 7 reviews related work that is relevant with the SC language system and our scheme for language extension. (Note that Chapters 4 to 6 also have their own related work sections.)

Finally Chapter 8 summarizes this thesis.

Chapter 2

The SC Language System

2.1 Overview

The *SC language system*, implemented in Common Lisp, deals with SC languages, which include

- SC-0, the base SC language, and
- extended SC-0 languages,

and consists of the following three kinds of modules:

- SC preprocessors—include SC files and handles macro definitions and expansions,
- SC translators—typically implemented by developers of language extension and transform an SC program into another SC program, and
- The SC compiler—compiles SC-0 code into C.

Figure 2.1 shows code translation phases in the SC language system. An extended SC code is transformed into SC-0 by the SC translators, then translated into C by the SC compiler. Before each transformation/translation is applied, preprocessing by an SC preprocessor is performed. As the figure shows, a series of translators can be applied one by one to get SC-0 code. Extension implementers write *transformation rule-sets* for the SC translators to transform the extended language into SC-0.

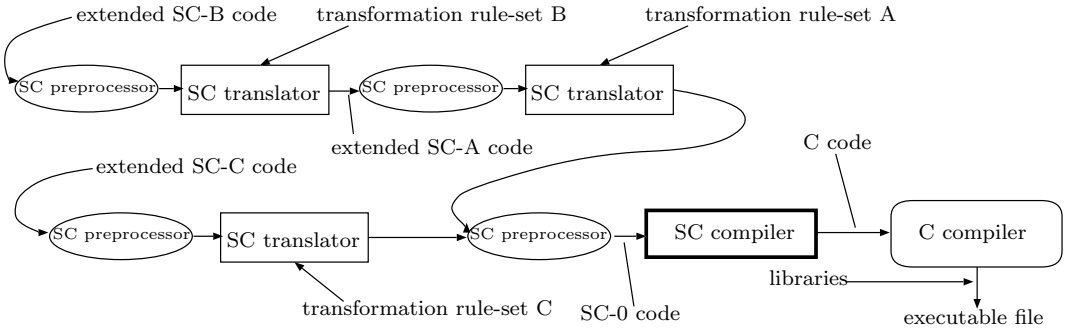


Figure 2.1: Code translation phases in the SC language system.

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

Figure 2.2: An SC-0 program.

```
int sum (int* a, int n)
{
  int s=0;
  int i=0;
  do{
   if (i >= n) break;
   s += a[i++];
  } while(1);
  return s;
}
```

Figure 2.3: C program equivalent to the SC-0 program in Figure 2.2.

2.2 The SC-0/1 Language and the SC Compiler

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

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

Figure 2.2 shows an example of such an SC-0 program, which is equivalent to the program in Figure 2.3.

Some features of C such as `->` operators, `for` constructs, and `while` constructs are not supported in SC-0 because they can be realized by simple transformation over S-expressions. The *SC-1* language, which is implemented as an SC translator into SC-0, features them.

The following sections describe the specification of the SC-1 language. The semantics is explained by showing equivalent C code fragments. See Appendix A for the complete syntax of SC-1.

2.2.1 Expressions

Table 2.1 shows examples of SC-1 expressions and C expressions equivalent to them. Arithmetic expressions and assignments are written in Lisp-like syntax using operators corresponding to ones of C. A list whose first element is not any keyword is evaluated as a function call. For conditional expressions, the keyword `if-exp` is used. (The keyword `if` cannot be used here, because expressions are distinguished from statements unlike Lisp.) The keyword `mref` is used for indirection and `ptr` for getting addresses. Like in Scheme, `aref` (array reference) is used for array subscripting, and `fref` (field reference) is used to get a member of structures or unions. Sizeof-expressions are written using `sizeof` operators. Cast-expressions need `cast` keywords unlike in C, to distinguish them from function calls explicitly. A "compound literal" of C99 [21] also can be written in SC-1 using the keyword `init`.

2.2.2 Statements

Table 2.2 illustrates SC-1 statements. Labeled-, selection-, iteration- and jump-statements are written like special forms of Scheme with corresponding keywords. A compound-statement is written as a list beginning with `begin`. An empty list `()` is evaluated as a null statement. The concept of statement-expressions is the same as in C.

2.2.3 Type expressions and type definitions

In SC-1, a type is expressed by a *type expression*. SC-1 provides the keywords which correspond to any basic types of C. The basic types expressed by multiple tokens

Table 2.1: SC-1 expressions.

C	SC-1
`x = 3+4*5`	`(= x (+ 3 (* 4 5)))`
`f(x, y[3][4], s->a.b)`	`(f x (aref y 3 4) (fref s -> a b))`
`x -= (int)(\&b) / *c`	`(-= x (/ (cast int (ptr b)) (mref c)))`
`(0<n && n<10)?a:b`	`(if-exp (and (< 0 n) (< n 10)) a b)`
`sizeof (a)`	`(sizeof a)`
`sizeof (int)`	`(sizeof int)`
`(funarray[3])(a, b)`	`((aref funarray 3) a b)`
`(struct sab){x, y}` (a compound literal)	`(init (struct sab) (struct x y))`

Table 2.2: SC-1 statements.

C	SC-1
`L1: goto L2;`	`(label L1 (goto L2))`
`;` (a null-statement)	`()`

`switch (n) {`
`  case 1: f(n); break;`
`  case 2: g(n); break;`
`  default: break;`
`}`
	`(switch n`
	`  (case 1) (f n) (break)`
	`  (case 2) (g n) (break)`
	`  (default) (break))`

`for (i=0 ; i<=n ; i++) {`
`  if (x<i)`
`  { x++; y++; }`
`  else return i;`
`  while (x<y) x*=2;`
`}`
	`(for ((= i 0) (<= i n) (inc i))`
	`  (if (< x i)`
	`  (begin (inc x) (inc y))`
	`  (return i))`
	`  (while (< x y) (*= x 2)))`

such as `unsigned int` are also expressed by a single symbol such as `unsigned-int` in SC-1. The syntaxes for enumerated, struct and union types are `(enum` *identifier*`)`, `(struct` *identifier*`)` and `(union` *identifier*`)` respectively.

For pointer types, the keyword `ptr` is used. For example, `(ptr int)` is the type of pointers to `int` values. The syntax for array types is

`(array` *element-type* $size_1$ ... $size_n$`)`,

where *element-type* is a type expression and each *size* is a constant expression. *Size*s are omitted for an array of unknown size (the size is determined by the initializer list). The syntax for function types is

`(fn` *return-type* $argument\text{-}type_1$ ... $argument\text{-}type_n$ $\texttt{va-arg}_{opt}$`)`,

where *return-type* and *argument-type*s are type expressions. `Va-arg` corresponds to an ellipsis (`, ...`) in C, which is used for functions which take variable arguments.

Any type expressions can be qualified by *type qualifiers*—`const`, `volatile` or `restrict`—in the following manner:

(*type-qualifier* ... *type-expression*).

For example, `(array (const int) 5)` is the type for arrays whose elements cannot be modified.

A type definition is written using the keyword `deftype` as follows:

`(deftype` *identifier type-expression*`)`.

SC-1 provides the separate syntax for type definitions while using a storage class specifier `typedef` in an ordinary declaration means a type definition in C.

Note that a single type expression includes complete information of a type. This is different from C in which type information is written separately in some cases. For example, a variable declaration of an array of `int` values is

```
int a[5];
```

where the element type and the array declarator are written separately. Its equivalent declaration of SC-1 is

```
(def a (array int 5)).
```

Such notation is more readable and easier to analyze. For an extreme example,

```
typedef void *(*(*gg_t)(void *(*) (int, int))) (int, int);
```

is much less readable than

```
(deftype gg-t (ptr (fn (ptr (fn (ptr void) int int))
                       (ptr (fn (ptr void) int int))))).
```

2.2.4 Declarations

The syntax for variable declarations/definitions is

(*storage-class-specifier identifier type-expression initializer*$_{opt}$).

A function definition/declaration is

(*storage-class-specifier* (*function-name parameter-name*$_1$ ... *parameter-name*$_n$)
(`fn` *return-type argument-type*$_1$ ... *argument-type*$_n$ `va-arg`$_{opt}$)
block-item ...)

in most cases. *Function-name* and *parameter-name*s are identifiers. *Argument-type*$_k$ ($1 < k < n$) specifies the type of *parameter-name*$_k$. Each *block-item* is a statement or a declaration/definition. *Storage-class-specifier* is a keyword which specifies a storage class (`static`, `extern` or `auto`) of the variable/function. The keyword `def` is also provided, which corresponds to omitting a storage-class specifier in C. In addition, the keywords `decl`, `static-decl`, `extern-decl` and `auto-decl` are provided, which are equivalent to `def`, `static`, `extern` and `auto` respectively except that the formers can be used for prototype declarations.

Initializer in a variable definition is an expression or a *compound initializer* for initializing an array or a struct. A compound initializer is a list which consists of `array` or `struct` and following initializers.

Table 2.3 illustrates SC-1 declarations/definitions and equivalent ones in C.

2.2.5 Definitions of enumerations, structs and unions

Enumeration constants are defined with the keyword `def`, an enum specifier which consists of an `enum` and an identifier, and following *enumerators*. Each enumerator is an identifier or a list of an identifier and a constant expression. The latter corresponds to an enumerator with `=` of C.

The structure of a struct/union is defined with a `def` or a `decl`, a struct/union specifier which consists of a `struct`/`union` and an identifier, and following member

Table 2.3: SC-1 declarations/definitions of variables/functions.

C	SC-1
`int x;`	`(def x int)`
`extern y;` (`extern int y;`)	`(extern y int)`
`static long z=x+1;`	`(static z long (+ x 1))`
`unsigned int a[]={1,2,3};`	`(def a (array unsigned-int) (array 1 2 3))`
`struct sab s={10,20};`	`(def s (struct sab) (struct 10 20))`
`int f(char x) { return x*x; }`	`(def (f x) (fn int char) (return (* x x)))`
`void f(void) {}`	`(def (f) (fn int void))`
`void f(void);`	`(decl (f) (fn int void))`
`extern void f(void);`	`(extern-decl (f) (fn int void))`

declarations. A bit-field can be specified by a `:bit` and a constant expression following a member declaration. The difference between `def` and `decl` is same as in a function declarations/definitions.

Unlike C, we cannot define an enumeration/struct/union and variables whose type is the being defined enumeration/struct/union at the same time. Therefore, an enumeration/struct/union cannot be unnamed in SC-1 because such an enumeration/struct/union is useless. By way of exception, we can use an unnamed enumeration/struct/union for a type definition as follows:

`(deftype` *identifier* `enum` *enumerator* ... `)`

`(deftype` *identifier struct-or-union member-declaration* ... `)`.

Table 2.4 illustrates SC-1 declarations/definitions of enumerations, structs and unions.

2.3 SC Preprocessors

SC preprocessors handle the following SC preprocessing directives, most of which correspond to the directives for the C preprocessor, to transform SC programs:

- `(%include` *file-name*`)`
  corresponds to an `#include` directive in C. The file *file-name* is included.

Table 2.4: SC-1 declarations/definitions of enumerations, structs and unions.

C	SC-1
`enum animal {CAT, DOG, OTHER=99};`	`(def (enum animal) CAT DOG (OTHER 99))`

`struct sab {`
`  int a;`
`  long b;`
`};`
	`(def (struct sab)`
	`  (def a int)`
	`  (def b long))`

`union byte {`
`  unsigned char n;`
`  struct {`
`  int n1:4;`
`  int n2:4;`
`  } n12;`
`}`
	`(def (struct s-n12)`
	`  (def n1 int) :bit 4`
	`  (def n2 int) :bit 4)`
	`(def (union byte)`
	`  (def n unsigned-char)`
	`  (def n12 (struct s-n12)))`

`typedef struct {`
`  char str[20];`
`} name_t;`
	`(deftype name-t struct`
	`  (def str (array char 20))`

`struct emp;`	`(decl (struct emp))`
`struct emp{};`	`(def (struct emp))`

- `(%defmacro` *macro-name lambda-list S-expression*$_1$ ... *S-expression*$_n$`)`
  evaluated as a `defmacro` form of Common Lisp to define an SC macro. After the definition, every list in the form of (*macro-name* ...) is replaced with the result of the application of Common Lisp's `macroexpand-1` function to the list. The algorithm to expand nested macro applications complies with the standard C specification.

- `(%defconstant` *macro-name S-expression*`)`
  defines an SC macro in the same way as a `%defmacro` directive, except that every symbol which `eq`s *macro-name* is replaced with *S-expression* after the definition.

- `(%undef` *macro-name*`)`
  undefines the specified macro defined by `%defmacro`s or `%defconstant`s.

- `(%ifdef` *symbol list*$_1$ *list*$_2$`)`

(`%ifndef` *symbol* $list_1$ $list_2$)
If the macro specified by *symbol* is defined, $list_1$ is spliced there. Otherwise $list_2$ is spliced.

- (`%if` *S-expression* $list_1$ $list_2$)
  S-expression is macro-expanded, and then the result is evaluated by Common Lisp. If the return value `eql`s `nil` or `0`, $list_2$ is spliced there. Otherwise $list_1$ is spliced.

- (`%error` *string*)
  interrupts the compilation with an error message *string*.

- (`%cinclude` *file-name*)
  file-name specifies a C header file. The C code is compiled into SC-0 and the result is spliced there. The SC programmers can use library functions and most of macros such as `printf` and `NULL` declared/`#defined` in C header files. The detail of this mechanism is discussed in Chapter 5.

2.4 SC Translators and Transformation Rules

A preprocessed SC program is analyzed and transformed into another SC program by an SC translator. Each transformation phase in Figure 2.1 is governed as a *transformation rule-set*, which is defined by some parameters and *transformation rules*. Each transformation rule is defined by pairs of a sequence of *patterns* and an action corresponding to it which is defined by Common Lisp forms, and the defined rule can be called as a Lisp function.

Rule-sets organize a hierarchical structure like classes of object oriented languages. That is, we can define a rule-set by extending pre-existing rule-sets.

The following sections show how to define and apply transformation rule(-set)s.

2.4.1 Defining rule-sets

The syntax for defining a rule-set is as follows:

(`define-ruleset` *rule-set-name* (*parent-rule-set-name* ...)

(*parameter-name*$_1$ *default-value*$_1$)

...

(*parameter-name*$_n$ *default-value*$_n$)).

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

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

- `entry`: The value must be a symbol and it specifies a rule which is called at the beginning when the rule-set is applied.

- `default-input-handler`: The value must be a function with one argument. This function is called if no pattern matches when a rule is applied.

2.4.2 Defining rules

Then we need to define transformation rules which belong to the rule-set:

(`defrule` *rule-name rule-set-name*

((`#?`*pattern*$_{11}$ ... `#?`*pattern*$_{1m_1}$) *form-list*$_1$)

...

((`#?`*pattern*$_{n1}$ ... `#?`*pattern*$_{nm_n}$) *form-list*$_n$)

[(`otherwise` *form-list*$_{otherwise}$)]

).[1]

When a rule is applied, the parameter is tested whether it is matched by any of pattern in the following order: *pattern*$_{11}$, ... *pattern*$_{1m_1}$, *pattern*$_{21}$, ... *pattern*$_{2m_1}$, ... *pattern*$_{n1}$, ... *pattern*$_{nm_n}$. The form list *form-list*$_i$ is evaluated (by a Lisp evaluator) when the argument is matched by *pattern*$_{ik}$. If no pattern matches the parameter, *form-list*$_{otherwise}$ is evaluated if exists, otherwise the function `default-input-handler` of the rule-set *rule-set-name* is called (the parameter is passed to the function).

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

We can define a rule using keyword `extendrule` instead of `defrule`. `Extendrule` has the same syntax and semantics as `defrule` except that the rule *rule-name* of a parent rule-set is applied when no pattern matches and no `otherwise` clause is given. In the case where a rule-set has multiple parent rule-sets, the applying order is determined by the "class precedence list" of CLOS (Common Lisp Object System) [42].

2.4.3 Patterns

We specify *pattern*s using notations similar to backquote macros. More precisely, *pattern* is an S-expression consisted of one of the following elements:

(1) *symbol*
matches a symbol that is `eq` to *symbol*.

(2) `,`*symbol*
matches any element.

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

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

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

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

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

to get an actual return value of *function* in (4) and (5) by passing the corresponding *symbol*, and the `call-next-rule` function can be used to call the applying rule of the next rule-set (like `call-next-method` of CLOS).

2.4.4 Applying rule-sets

A defined rule-set (transformation phase) can be applied using the function `apply-ruleset`:
`(apply-ruleset` *input* `:`*rule-set-name*`)`.
The `apply-ruleset` receives additional keyword parameters corresponding to *parameter-name*s, which change values of rule-set parameters. Here shows an example:

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

This function applies the `expression` rule of the specified rule-set `sc0-to-sc0` to *input*.[2]
In processing `apply-ruleset`, dynamic variable `*current-ruleset*` which stands for the *current rule-set* is dynamically bound to the specified rule-set.

2.4.5 Applying rules

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

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

2.4.6 Example

Figure 2.4 shows an example of definitions of transformation rule-sets. A character '`~`' in this figure is a macro character which works as if it were a backquote, except that symbols in the following expression are interned to a distinguished package for

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

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

(defrule block-item sc0-to-sc0
  ((#?,bi[declaration]
    #?,bi[statement])
   (get-retval 'bi))
  )
(defrule statement sc0-to-sc0
  (#?(do-while ,exp ,@body)
   ~(do-while ,(expression exp)
      ,@(mapcar #'block-item body)))
  ...
  (otherwise (expression x)) ;expression-statements
  )
...

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

Figure 2.4: Transformation rule-sets.

SC code instead of the current package (`*package*`) [3]. Two rule-sets are defined in this figure. The `sc0-to-sc0` rule-set defines an identical transformer from SC-0 to SC-0. The `sc1-to-sc0` rule-set defines a transformation from the SC-1 language to SC-0. SC-1 features several constructs for iteration and bindings added to SC-0.

Suppose that the form

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

is evaluated. This evaluation is progressed as follows:

1. The `statement` rule of `sc1-to-sc0` is applied to the given input; no pattern matches.

2. The `statement` rule of `sc0-to-sc0` is applied; the pattern `#?(do-while ...)` matches.

3. Because the `expression` rule of `sc1-to-sc0` is not defined, the `expression` rule of `sc0-to-sc0` is applied to `x`; `x` is returned. (the `expression` rule is snipped in the figure.)

4. Because the `block-item` rule of `sc1-to-sc0` is not defined, the `block-item` rule of `sc0-to-sc0` is applied to `(while y (++ z))`, and then the `statement` rule of `sc1-to-sc0` is applied; `(if y (do-while y (++ z)))` is returned.

5. `(do-while x (if y (do-while y (++ z))))` is returned.

Since the current rule-set varies according to context, the rule of the current rule-set also varies. For example, when

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

is evaluated, the `statement` rule of `sc0-to-sc0` is applied to `(while ...)` and `no-match-error` occurs.

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

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

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

Figure 2.5: Implementation code for defining rule-sets.

2.5 Implementation

The rule-set definition facility is implemented using CLOS. Figure 2.5 shows (simplified) implementation code for `define-ruleset`, `defrule`, `extendrule` and `apply-ruleset`. Roughly speaking, `define-ruleset` and `defrule` (or `extendrule`) correspond to `defclass` and `defmethod` respectively so that we can dispatch an appropriate rule for the being applied rule-set.

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

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

rule-set argument in the definitions.

The SC compiler is also implemented as a transformation rule-set, which specifies transformation from S-expressions as an SC-0 program to a string (instead of S-expressions) as a C program.

Chapter 3

Evaluation and Discussion

3.1 Implementation Cost for Language Extension

To evaluate implementation cost using our system, we actually implemented a simple language extension to SC-0 and Cilk [10], and compared the implementation costs. Cilk, an extended C language with fine-grained multi-threading, is implemented by source-to-source translation. In this evaluation, we implemented the extension by modifying the translator.

3.1.1 The extended language for evaluation

For evaluation, we implemented labeled-`breaks`/`continues` as an extension. Figure 3.1 and Figure 3.2 show program examples of extended SC-0 and Cilk respectively. In the case of Figure 3.2, the labeled break statement `break L1` breaks out of the outer and the inner `do-while` loop immediately, and the labeled continue statement `continue L1` breaks out of the inner loop and iterate on the outer loop.

3.1.2 Implementation strategy

Labeled-`breaks`/`continues` can be implemented by translating them into `goto` statements and putting additional labels before and after the labeled-statement. For example, the program in Figure 3.2 is translated to the program in Figure 3.3. (The program in Figure 3.1 is also translated in the same manner.)

```
(def (f x) (fn int int)
  (label L1
   (do-while (> x 0)
     (do-while (<= x 10)
       (inc x)
       (if (< x 5)
           (continue L1)
           (break L1))
       (= x (g x)))
     (= x (h x))))
  (return x))
```

Figure 3.1: A labeled `break`/`continue` (in SC-0).

```
int f( int x )
{
L1:
  do{
    do{
      x++;
      if( x>5 ) continue L1;
       else  break L1;
      x = g(x);
    } while (x <= 10);
    x = h(x);
  } while (x > 0);
  return x;
}
```

Figure 3.2: A labeled `break`/`continue` (in Cilk).

```
int
f (int x)
{
L1:
  do{
    do{
      x++;
      if (x < 5)  goto L12;
       else goto L13;
      x = g (x);
    } while (x <= 10);
    x = h (x);
    L12: ;
  } while (x > 0);
L13: ;
  return x;
}
```

Figure 3.3: Implementation of labeled `break` and `continue` statements in C.

```
(defvar *label-list*)
(define-ruleset labeled-break (sc0-to-sc0))

(defrule sc-program labeled-break
  (#?(,@declaration-list)
   (let ((*label-list* '()))
     (call-next-rule))))

(defrule statement labeled-break
  ((#?(label ,id (do-while ,exp ,@body))
    #?(label ,id (switch ,exp ,@body)))
   (let ((*label-list* (cons (list id (generate-id "cont")
                                   (generate-id "break"))
                             *label-list*)))
     ~(begin (label ,id
                    (,(car (third x)) ,exp ,@(sc-block body)
                      (label ,(second (first *label-list*)) nil)))
             (label ,(third (first *label-list*)) nil))) )
  ((#?(continue ,id)
    #?(break ,id))
   (let ((label-tuple
          (car (member id *label-list* :key #'first))))
     (unless label-tuple
       (error "label ~s is undefined." id))
     ~(goto ,(funcall (case (car x)
                        ((continue) #'second)
                        ((break) #'third))
                      label-tuple)))) )
```

Figure 3.4: The transformation rule for labeled `break` and `continue` statements.

3.1.3 Comparison of the implementation costs

We implemented the extension to SC-0 by the transformation rule-set shown in Figure 3.4.

The same extension to Cilk was implemented by modifying the original Cilk-to-C translator, which first translates source code into an AST (Abstract Syntax Tree), applies analysis and transformation to it, and then generates C code. We had to not only modify the code for AST transformation, but also define an additional structure for AST and modify the parser (`.yacc` files).

The comparison of implementation costs based on the number of the modified lines and files is shown in Table 3.1. We had to modify more lines and files for Cilk. The number of modified files for Cilk indicates that the changes of syntax required modifying the multiple components of the translator. The cost for finding out all code fragments

Table 3.1: Comparison of implementation costs for labeled `break`s and `continue`s.

	# of files	# of lines
SC	1	45
Cilk	9	112

which should be modified is considerable.

3.2 Discussion

In the earlier version of the SC system, the definition of transformation rules which correspond to Figure 2.4 can be written as in Figure 3.5. All the rules were at the same level and the concept of "rule-set" did not exist explicitly. (Though we used this term, which simply indicated a group of rules which was related to one transformation phase.) Therefore we had to write rules for the whole syntax, even if an extended language has only several new constructs.

3.2.1 Extensibility of rule-sets

In the current version, existing rule-sets can be extended to define a new rule-set. In Figure 2.4, the `sc1-to-sc0` rule-set is defined by extending `sc0-to-sc0`, which is a rule-set for identical transformation. By using `sc0-to-sc0` as a common template, many transformation rule-sets can be defined only by describing the difference.

Extending rule-sets is also helpful when we use a commonly-used rule-set as part of the entire transformation. For instance, when we implement high-level services, we divide the entire transformation into several phases. Indeed, when we implemented MT-SC, SC-0 with multi-threading (Section 4.6.2), we used the `type-info` rule-set (Section 4.4.3) to add type information to all expressions. In this case we cannot use the original `type-info` rule-set as it is, but we can easily extend it for MT-SC (See Figure 3.6). In a similar manner, we can easily reuse the `type-info` rule-set for implementing other extended languages by writing a small amount of additional code.

In addition, extending rule-sets is also helpful when we want to use multiple extensions

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

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

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

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

Figure 3.5: The earlier version of transformation rules.

at the same time. Suppose we have already implemented extensions A and B, and we want an extended language which features both of them. As Figure 3.7 shows, such an implementation could be done by connecting two rule-sets serially one after another. If we choose the left path, we must modify the rule-set for B to support new features added in A. This modification can be done by the rule-set extension facility described above. We can write the difference separately instead of rewriting the rule-set B. This scheme works well if A is a relatively simple extension (e.g. new constructs for iteration). However if both A and B are non-trivial extensions, extending a rule-set is not easy because the

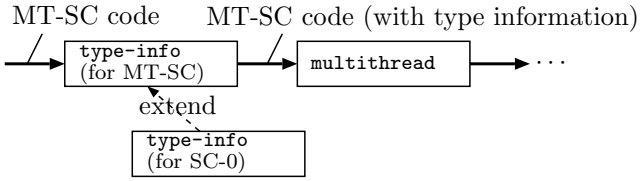


Figure 3.6: A rule-set as part of the entire transformation.

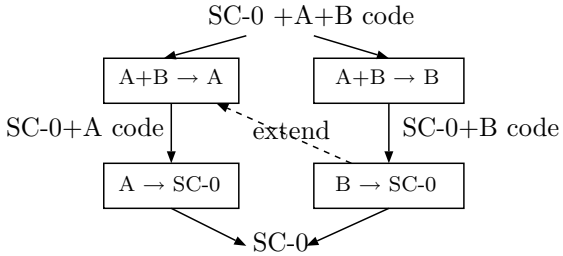


Figure 3.7: Applying multiple rule-sets.

semantics for A+B is not straightforward.

Figure 3.8 shows code for extending the `sc1-to-sc0` rule-set to fit MT-SC. Because such differential code can be managed independently of base rule-sets, we can manage rule-sets more clearly.

3.2.2 Ease of use

In the old version, we used to introduce more declarative notation for transformation rules in consideration of other rule-based languages such as Prolog. Now transformation rules are described with ML-like notations for pattern-matching, mainly because of implementation convenience. Some people may prefer the old syntax, but we think the new one is more intuitive because rules look like BNF notations.

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

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

Figure 3.8: Extending the `sc1-to-sc0` rule-set for MT-SC.

Chapter 4

Transformation-based Implementation of Lightweight Nested Functions

This chapter shows a practical example of implementation by language extension using the SC language system. This extension is not just an example of language extension; it extends the facility of the SC language system.

4.1 Introduction

The SC language system allows us to implement language extensions to C as code transformation at low cost. The fact that C has low-level operations (e.g., pointer operations) enables us to implement many flexible extensions using the SC language system. But without taking "memory" addresses, C lacks an ability to access variables sleeping in the execution stack, which is required to implement high-level services with "stack walk" such as capturing a stack state for check-pointing and scanning roots for copying GC (Garbage Collection).

A possible solution to this problem is to support *nested functions*. A nested function is a function defined inside another function. Each function can manipulate its caller's local variables (or local variables of its indirect callers) sleeping in the execution stack by indirectly calling a nested function of its (indirect) caller.

This chapter presents the implementation of an extended SC language, named LW-SC (LightWeight SC), which features nested functions. Many high-level services with "stack walk" mentioned above can be easily and elegantly implemented by using LW-SC as an intermediate language. Moreover, such services can be efficiently implemented because we design and implement LW-SC to provide "lightweight" nested functions by aggressively reducing the costs of creating and maintaining nested functions. Though the GNU C compiler [41] (GCC) also provides nested functions as an extension to C, our elaborated translator to standard C is more portable and efficient for occasional "stack walk".

Note that, though this chapter presents an implementation using the SC language system, our technique is not limited to it.

4.2 Language Specification of LW-SC

LW-SC has the following features as extensions to SC-1.

- **Nested function types:**
  (`lightweight` *return-type argument-type*$_1$ ... *argument-type*$_n$ `va-arg`$_{opt}$)
  is added to the syntax for type expressions.

- **Calling nested functions:** In function-call expressions, (*expression* ...), the type of the first expression can be a nested function pointer type in addition to an ordinary function pointer type.

- **Defining nested functions:** Nested functions can be defined with the following form:
  (*storage-class-specifier* (*function-name parameter-name*$_1$ ... *parameter-name*$_n$)
  (`lightweight` *return-type argument-type*$_1$ ... *argument-type*$_n$ `va-arg`$_{opt}$)
  block-item ...)
  Note that this form is similar to that of ordinary function definitions. The only difference is that we use `lightweight` rather than `fn` for nested functions. Although ordinary functions can be defined at the top level, nested function can be defined wherever non-top-level variable definitions can appear.

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

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

(def (main) (fn int)
  (return (foo 1))
```

Figure 4.1: A LW-SC program.

A nested function can access the lexically-scoped variables in the creation-time environment and a pointer to it can be used as a function pointer to call the closure. Figure 4.1 shows an LW-SC program. When `h` indirectly calls the nested function `g1`, it can access the parameter `a` and the local variables `x`, `y` sleeping in `foo`'s frame.

4.3 GCC's Implementation of Nested Functions

GCC also features nested functions and the specification of nested functions in LW-SC is almost the same as the one of GCC. Unlike closure objects in modern languages such as Lisp and ML, nested functions of both GCC and LW-SC are valid only when the function frame of the enclosing function still remains on the stack. In addition, unlike GCC, pointers to nested functions of LW-SC are not compatible with those to top-level functions.[1] However, such limitations are insignificant for the purpose of implementing most of high-level services with "stack walk" mentioned in Section 4.1.

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

- The cost of initializing a nested function is high. When initializing a nested function, GCC uses a technique called *trampolines* [3] in order to obtain the address

[1] That is, a variable of a nested function pointer type cannot be assigned to any top-level function pointer (and vice versa).

of the function. Trampolines are code fragments generated on the stack at runtime. It is used to enter the nested function with a necessary environment. The cost of runtime code generation is high, and for some processors like SPARC, it is necessary to flush an instruction cache for the runtime-generated trampoline code.

- Local variables and parameters of a function generally may be assigned to registers if the function has no nested function. But an owner function of GCC's nested functions must keep the values of these variables in the stack since the nested functions may also access them. Thus, the owner function must perform memory operations to access these variables, which means that the cost of maintaining nested functions is high.

LW-SC reduces the first cost by translating the nested function to a lazily-initialized pair (on the explicit stack) of the ordinary function pointer and the frame pointer, and the second cost by saving the local variables to the "explicit stack" lazily (only on calls to nested functions), as is shown in the next section.

4.4 Implementation of LW-SC

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

4.4.1 Basic ideas

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

- After transformation, all definitions of nested functions are moved to be top-level definitions.

- To enable the nested functions to access local variables of their owner functions, an explicit stack is employed in C other than the (implicit) execution stack for C. The explicit stack mirrors values of local variables in the execution stack, and is referred to when local variables of the owner functions are accessed.

- To reduce costs of creating and maintaining nested functions, operations to fix inconsistency between two stacks are delayed until nested functions are actually invoked.

Function calls/returns and function definitions in LW-SC should be appropriately transformed based on these ideas.

4.4.2 Transformation strategy

LW-SC programs are translated in the following way to realize the ideas described in Section 4.4.1.

(a) Each generated C program employs an explicit stack mentioned above on memory. This shows a logical execution stack, which manages local variables, callee frame pointers, arguments, return values of nested functions (of LW-SC) and return addresses.

(b) Each function call to an ordinary top-level function in LW-SC is transformed to the same function call in C, except that a special argument is added which saves the stack pointer to the explicit stack. The callee first initializes its frame pointer with the stack pointer, moves the stack pointer by its frame size, then executes its body.

(c) Each nested function definition in LW-SC is moved to the top-level in C. Instead, a value of a structure type, which contains the pointer to the moved nested function and the frame pointer of the owner function, is stored on the explicit stack. Note that initialization of the structure is delayed until nested functions are invoked to reduce costs of creating nested functions.

(d) Each function call to a nested function in LW-SC is translated into the following steps.

    1. Push arguments passed to the nested function and the pointer to the structure mentioned above in (c) to the explicit stack.

2. Save the values of the all local variables and parameters, and an integer corresponding to the current execution point (return address) into the explicit stack, then return from the function.

3. Iterate Step 2 until control is returned to `main`. The values of local variables and parameters of `main` are also stored to the explicit stack.

4. Referring to the structure which is pointed to by the pointer pushed at Step 1 (the one in (c)), call the nested function whose definition has been moved to the top-level in C. The callee first obtains its arguments by popping the values pushed at Step 1, then executes its body.

5. Before returning from the nested function, push the return value to the explicit stack.

6. Reconstruct the execution stack by restoring the local variables, the parameters, and the execution points, with the values saved in the explicit stack at Step 3 (the values may be changed during the call to the nested function), to return to (resume) the caller of the nested function.

7. If necessary, get the return value of the nested function pushed at Step 5 by popping the explicit stack.

Note that a callee (a nested function) can access the local variables of its owner functions through the frame pointers contained in the structure that have been saved at Step 1.

For example, Figure 4.2 shows the state transition of the two stacks[2], in the case of Figure 4.1, from the beginning of the execution until the end of the first indirect call to a nested function `g1` (Each number in the figure corresponds to the step of the nested function call described in (d)). Notice that the correct values of the local variables are saved in the explicit stack during the execution of the nested function and otherwise in the C stack.

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

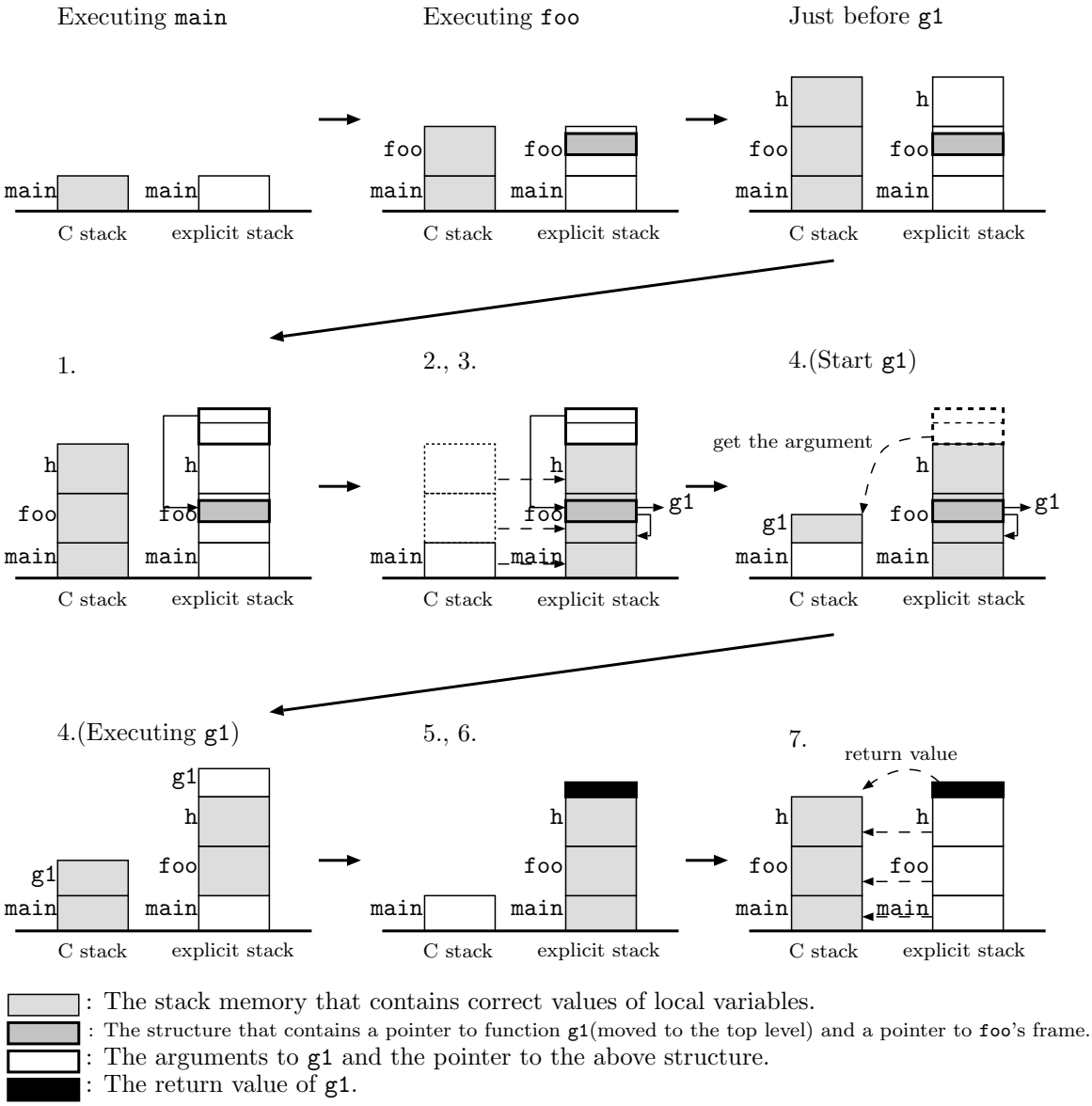


Figure 4.2: Details of an indirect call to the nested function `g1` in Figure 4.1.

4.4.3 Transformation rule-sets

To implement the transformation described above, we wrote transformation rules. The entire transformation is divided into the following five phases (rule-sets) for simplicity

and reusability of each phase.

(1) **The `lw-sc1` rule-set:** removes SC-1 extensions such as `while` statements by transforming them into equivalent SC-0 code fragments.

(2) **The `lw-type` rule-set:** adds type information to all the *expression*s of an input program.

(3) **The `lw-temp` rule-set:** transforms an input program in such a way that no function call appears as a subexpression (except as a right hand side of an assignment).

(4) **The `lightweight` rule-set:** performs the transformation described in Section 4.4.2.

(5) **The `untype` rule-set:** removes the type information added by the `type` rule-set from *expression*s to generate correct SC-0 code.

The details of these transformation rule-sets are presented below.

The `lw-sc1` rule-set As described in Section 2.2, some features of SC-1, which are also featured in LW-SC, are implemented by transformation into SC-0. The `lw-sc1` rule-set applies the same transformation to an LW-SC program, so to say, transforms it into an "LW-SC-0" program. By applying such transformation at the beginning, following rule-set can be written with fewer patterns.

We can write this rule-set easily by extending the `sc1-to-sc0` rule-set which the SC language system provides. All we have to write is:

```
(define-ruleset lw-sc1 (sc1-to-sc0))
(extendrule function-type lw-sc1
  (#?(lightweight ,@t-exprs)
   ~(lightweight ,@(mapcar #'type-expression t-exprs))))
```

to enable the translator to accept type expressions in the additional syntax. [3]

[3]Since function definitions in another function are syntactically correct in standard C [21], the original `sc1-to-sc0` translator also accepts them.

```
(def (g x) (fn int int)
  (return (* x x)))

(def (f x) (fn double double)
  (return (+ x x)))

(def (h x) (fn char double)
  (return (f (g x))))
```

Figure 4.3: An example for the `lw-type` rule-set (before transformation).

```
(def (g x) (fn int int)
  (return (the int
               (* (the int x) (the int x)))))

(def (f x) (fn double double)
  (return (the double
               (+ (the double x) (the double x)))))

(def (h x) (fn char double)
  (return (the double
              (call (the (fn double double) f)
                    (the int (call (the (fn int int) g)
                                   (the double x)))))))
```

Figure 4.4: An example for the `lw-type` rule-set (after transformation).

The `lw-type` rule-set Transformation by the `temp` rule-set and the `lightweight` rule-set needs type information of all expressions. The `lw-type` rule-set adds such information. More concretely, it transforms each *expression* into `(the` *type-expression expression*`)`. In addition, this rule-set adds a `call` into each function-call expression for convenience. For example, the program in Figure 4.3 is transformed to the program in Figure 4.4 by this rule-set.

Figure 4.5 shows the (abbreviated) transformation rule-set. The `lw-type` rule-set is defined by extending the `type-info` rule-set, which defines a transformer for SC-0. As well as `lw-sc1`, `lw-type` is adjusted to LW-SC by extending the `function-type` rule.

The translator manages environments to save the information about defined variables, structures, etc. using the dynamic variable `*env*`. The `type:add-variable` function renews the environment and a new environment is created at `begin` statements etc. The extended `expression` rule adds type information to expressions, referring to the dynamic variable by the `type:get-variable` function.

```
(define-ruleset type-info (sc0-to-sc0))
(defrule declaration type-info
  ;; variable declarations/definitions
  ((#?(,scs[storage-class-specifier] ,id[#'atom] ,texp ,@init))
   (let ((texp-ret (type-expression texp)))
     (type:add-variable id texp-ret)
     ~(,scs ,id ,texp-ret ,@(mapcar #'initializer init)) ))
  ;; function declarations/definitions
  (#?(,scs[storage-class-specifier] (,@id-list) ,ftype[function-type] ,@body)
   (type:add-variable (first id-list) ~(ptr ,(get-retval ftype)))
   (let ((*env* (type:make-environment *env*)))
     (mapc #'type:add-variable (cdr id-list) (cddr (get-retval ftype)))
     ~(,scs (,@id-list) ,(get-retval ftype) ,@(sc-block body)) ))
  ...)

;;; Called in processing begin, do-while, etc.
(defrule sc-block type
  (#?(,@bi-list)
   (let ((*env* (type:make-environment *env*))) ; addition
     (mapcar #'block-item bi-list))))

(defrule expression type-info
  ;; identifiers
  (#?,id[identifier]
   ~(the ,(type:get-vartype id *env*) ,id))
  ;; constants
  ((#?,num[#'integerp])
   ~(the int ,num) )
  ...
  ;; operators
  (#?(,op[comparator] ,@expr-list)
   ~(the int (,op ,@(mapcar #'expression expr-list))))
  (#?(,op[arithmetic-operator] ,@expr-list)
   (let* ((t-exprs (mapcar #'expression expr-list))
          (types (mapcar #'second t-exprs)))
     ~(the ,(reduce #'type-conversion types) (,op ,@t-exprs))))
  (#?(ptr ,expr)   ; the & operator
   (let* ((t-expr (expression expr))
          (tp (second t-expr)))
     ~(the (ptr ,tp) (ptr ,t-expr))))
  ...
  ;; function calls
  (#?(,fexp[expression] ,@expr-list)
   (let ((t-exprs (mapcar #'expression (cons fexp expr-list)))
         (func-type (second (car t-exprs))))
     ~(the ,(return-type func-type)
           (call ,@t-exprs)))))

;;;; extend the type rule-set
(define-ruleset lw-type (type-info))

;;; accept lightweight in addition to fn
(extendrule function-type lw-type
  (#?(lightweight ,@t-exprs)
   ~(lightweight ,@(mapcar #'type-expression t-exprs))))
```

Figure 4.5: The `lw-type` rule-set (abbreviated).

```
(def (g x) (fn int int)
  ;; (return (+ (= x 3) (g x)))
  (return (the int (+ (the int (= (the int x) (the int 3)))
                      (the int (call (the (fn int int) g)
                                     (the int x)))))))
```

Figure 4.6: An example for the `lw-temp` rule-set (before transformation).

```
(def (g x) (fn int int)
  (def tmp1 int)
  (def tmp2 int)
  ;; (= tmp1 (= x 3))
  (the int (= (the int tmp1)
              (the int (= (the int x) (the int 3)))))
  ;; (= tmp2 (g x))
  (the int (= (the int tmp2)
              (the int (call (the (fn int int) g)
                             (the int x)))))
  ;; (return (+ tmp1 tmp2))
  (return (the int (+ (the int tmp1) (the int tmp2)))))
```

Figure 4.7: An example for the `lw-temp` rule-set (after transformation).

The `lw-temp` rule-set A function call appearing as a subexpression such as `(g x)` in `(f (g x))` makes it difficult to add some operations just before/after the function call. The `lw-temp` rule-set makes such function calls not appear. Because some temporary variables are needed for the transformation of —lw-temp—, the definitions of those are inserted at the head of the function body. For example, the program in Figure 4.6 is transformed to the program in Figure 4.7 by this rule-set, where the definitions of `tmp1` and `tmp2` are added.

Figure 4.8 shows the (abbreviated) `lw-temp` rule-set. As well as `lw-type`, `lw-temp` is defined by extending the `temp` rule-set, which is defined by further extending the `sc0t-to-sc0t` rule-set. `Sc0t-to-sc0t`, which defines an identical transformer, is almost the same as `sc0-to-sc0` except that it accepts expressions with type information (in the `(the` *type-expression expression*`)` syntax). This rule-set can be written easily by extending `sc0-to-sc0`.

The actual transformation is performed by the `expression` rule, which returns an expression as its return value and causes the following side-effects:

```
(define-ruleset temp (sc0t-to-sc0t))

(extendrule declaration temp
  ;; function declarations/definitions
  (#?(,scs[storage-class-specifier] (,@id-list) ,ftype[function-type] ,@body)
   (let* ((*additional-defs* ())
          (new-body (sc-block body)))
     ~(,scs (,@id-list) ,ftype
         ,@*additional-defs* ,@new-body)))
  ...)

(defrule statement temp
  ;; Do-while needs to be transformed so that *additional-stmts*
  ;; are evaluated before each iteration even if after (continue).
  (#?(do-while ,expr ,@body)
   (let ((*preceding-stmts* ()))
     (let ((expr-ret (expression expr))
           (body-ret (sc-block body))
           (end-label (generate-id "loop_end")))
       ~(begin
          (do-while 1
            ,@*preceding-stmts*
            (if (the int (not ,expr-ret)) (goto ,end-label))
            ,@body-ret)
          (label ,end-label ())))))
  ;; expression-statements
  (#?(the ,@rest)
   (let ((*preceding-stmts* ()))
     (let ((ret (expression x)))
       ~(begin ,@*preceding-stmts* ,ret))))
  (otherwise
   (let ((*preceding-stmts* ()))
     (let ((ret (call-next-rule)))  ;call statement of the parent rule-set.
       ~(begin ,@*preceding-stmts* ,ret))))
  )

(defrule expression temp
  ...
  ;; function calls
  (#?(the ,texpr (call (the ,ftexpr ,fexpr) ,@arg-list))
   (let* ((the-fexpr ~(the ,ftexpr ,fexpr))
          (exps-ret (mapcar #'expression (cons the-fexpr arg-list)))
          (call-expr ~(the ,texpr (call ,@exps-ret))))
     (if (void-p texpr)
         (progn
           (temp:add-precedent call-expr)
           ~(the int 0))
         (let* ((tmp-id (generate-id "tmp")) ;make a fresh variable
                (tmp-expr ~(the ,texpr ,tmp-id)))
           (temp:add-variable tmp-id texpr)
           (temp:add-precedent ~(the ,texpr (= ,tmp-expr ,call-expr)))
           tmp-expr)) ))
  ...)

;;;; extend the type rule-set
(define-ruleset lw-temp (temp))

;;; accept lightweight in addition to fn
(extendrule function-type lw-temp
  (#?(lightweight ,@t-exprs)
   ~(lightweight ,@(mapcar #'type-expression t-exprs))))
```

Figure 4.8: The `lw-temp` rule-set (abbreviated).

- adding the variable definitions to be inserted at the head of the current function (such as `(def tmp1 int)` and `(def tmp2 int)` in Figure 4.6) to the dynamic variable `*additional-defs*`, and
- adding the assignments to be inserted just before the expression (`(= tmp1 (= x 3))` and `(= tmp2 (g x))`) to `*preceding-stmts*`.

The definitions saved in `*additional-defs*` are inserted by the `declaration` rule and the assignments in `*preceding-stmts*` are inserted by the `statement` rule.

The `lightweight` rule-set Now the transformation described in Section 4.4.2 is realized by the `lightweight` rule-set. Figure 4.9 shows the (abbreviated) `lightweight` rule-set which is related to the transformation of "ordinary function" calls and "nested function" calls. `Esp` appearing in the code is a special parameter which is added to each function and keeps the stack top of the explicit stack. `Efp` is a special local variable added to each function, which acts as the (explicit) frame pointer of the function. `Lwe-xfp` transforms references to local variables into references to the explicit stack.

"Ordinary function" calls and "nested function" calls can be statically distinguished with the functions' types because ordinary function types are incompatible with lightweight nested function types.

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

Calling ordinary functions: The function call is performed as a part of the conditional expression of the `while` statement, where the stack pointer is passed to the callee as an additional first argument. If the callee procedure normally finished, the condition becomes false and the body of `while` loop is not executed. Otherwise, if the callee returned for a "nested function" call, the condition becomes true. In the body of the `while` loop, the values of local variables are saved to the explicit stack, an integer that corresponds to the current execution point is also saved to the explicit stack (`(fref efp -> call-id)`), and then the current function temporarily exits. This function is re-called for reconstructing the execution stack after the execution of the nested function. Then the control is transferred to the `label` that

```
(define-ruleset lightweight (sc0t-to-sc0t))

;;;; Due to the lw-temp rule-set, a function call expression must be appeared in either of the following form
;;;; as a statement expression:
;;;;  * (= variable function-call-expression)
;;;;  * (= function-call-expression).
(defrule expression lightweight
  ;; "Ordinary function" calls
  ((#?(the ,texp0 (= (the ,texp1 ,id) (the ,texp (call (the (fn ,@texprs) ,exp-f) ,@exprs))))
    #?                                (the ,texp (call (the (fn ,@texprs) ,exp-f) ,@exprs)))
   (let* (...)
     Adds local variable definitions etc.
     ~(begin
        (= new-esp esp)
        ...
        (while (and (== (= ,(Lwe-xfp `(the ,texp1 ,id))
                           (call ,fexp new-esp ,@(cdr tmpid-list)))
                        (special ,texp0))
                    (!= (= (fref efp -> tmp-esp) (mref-t (ptr char) esp))))
          ;; Save the values of local variables to the frame.
          ,@(make-frame-save *current-func*)
          ...
          ;; Save the current execution point.
          (= (fref efp -> call-id)
             ,(length (finfo-label-list *current-func*)))
          ;; Return from the current function
          ;; (In main, call the nested function here instead of the following steps).
          ,(make-suspend-return *current-func*)
          ;; Continue the execution from here when reconstructing the execution stack.
          (label ,(caar (push (cons (generate-id "L_call" *used-id-list*) nil)
                              (finfo-label-list *current-func*)))
                 nil)
          ;; Restore local variables from the explicit stack.
          ,@(make-frame-resume *current-func*)
          ...
          (= new-esp (+ esp 1))))))
  ;; "Nested function" calls
  ((#?(the ,texp0 (= (the ,texp1 ,id) (the ,texp (call (the (lightweight ,@texprs) ,exp-f) ,@exprs))))
    #?                                (the ,texp (call (the (lightweight ,@texprs) ,exp-f) ,@exprs)))
   (let* (...)
     Adds local variable definitions etc.
     ~(begin
       ...
       (= argp (aligned-add esp (sizeof (ptr char))))
       ;; Push the arguments passed to the nested function
       ,@(mapcar (compose #'(lambda (x) `(push-arg ,(second x) ,(third x) argp))
                          #'Lwe-xfp)
                 (reverse exprs))
       ;; Push the structure object that corresponds to the frame of the nested function to the explicit stack.
       (= (mref-t (ptr closure-t) argp) ,xfp-exp-f)
       ...
       ;; Save the values of local variables to the frame.
       ,@(make-frame-save *current-func*)
       (= (fref efp -> argp) argp)
       (= (fref efp -> tmp-esp) argp)
       ;; Save the current execution point.
       (= (fref efp -> call-id)
          ,(length (finfo-label-list *current-func*)))
       ;; Return from the current function (In main, call the nested function here instead of the following steps).
       ,(make-suspend-return *current-func*)
       ;; Continue the execution from here after the function call finishes.
       (label ,(caar (push (cons (generate-id "L_call" *used-id-list*) nil)
                           (finfo-label-list *current-func*)))
              nil)
       ;; Restore local variables from the explicit stack.
       ,@(make-frame-resume *current-func*)
       ;; Get the return value (if necessary).
       ,@(when assign-p
           `( (= ,(Lwe-xfp `(the ,texp1 ,id))
                 (mref-t ,texp1 (fref efp -> argp))) )) ))))
```

Figure 4.9: The `lightweight` rule-set (abbreviated).

is put next to the `return` by a `goto` statement which is added in the head of the function. Then the values of local variables are restored form the explicit stack and the function call in the conditional expression of the `while` statement is restarted. The assignment `(= new-esp (+ esp 1))` at the end of the `while` block sets a flag at the LSB of the explicit stack pointer that indicates reconstructing the execution stack.

Calling Nested functions: The arguments passed to the nested function and the closure structure (contains the nested function pointer and the frame pointer of its owner function) are pushed to the explicit stack. Then, like an "ordinary function" call, the values of local variables and the executing point are saved, the current function exits, and the execution point is restored by `goto` after the procedures for calling the nested function. Then the values of local variables are restored and the return value of the nested function is taken from the top of the explicit stack, if exists.

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

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

- initializing the frame pointer of the explicit stack (`efp`) and the stack pointer (`esp`),
- judging whether reconstruction of the execution stack is required or not and, if required, executing `goto` to the `label` corresponding to `(fref efp -> call-id)`, and
- popping parameters from the explicit stack, in the case of nested functions.

The transformation also involves adding the parameter `esp` that receives the explicit stack pointer, adding some local variable definitions, and adding the structure

```
(define-ruleset untype ()
  (entry 'iter))

(defrule iter untype
  ((#?(the ,texp ,exp))
   (iter exp))
  ((#?(call ,@exp-list))
   (mapcar #'iter exp-list))
  ((#?(,@lst))
   (mapcar #'iter lst))
  (otherwise x))
```

Figure 4.10: The `untype` rule-set.

definition that represents the function's frame in the explicit stack and is referred to by `efp`.

The `untype` rule-set The output code transformed by the `lightweight` rule-set is not valid SC-1 code because it contains type information. The `untype` rule-set removes such information and generate valid SC-1 code. The rule-set is very simple; only needs to search `(the ...)` forms recursively and to remove the type information. Figure 4.10 shows the entire `untype` rule-set.

As an example of the total translation, Appendix B shows the entire SC-1 code generated from the LW-SC program in Figure 4.1.

4.5 Evaluation

4.5.1 Creation and maintenance cost

To measure costs of creating and maintaining nested functions, we employed the following programs with nested functions for several high-level services and compared them with the corresponding plain C programs:

BinTree (copying GC) creates a binary search tree with 200,000 nodes, with a copying-collected heap (Figure 4.11).

```
(deftype sht (ptr (lightweight void void)))
(def (randinsert scan0 this n)
     (fn void sht (ptr Bintree) int)
  (decl i int)
  (decl k int)
  (decl seed (array unsigned-short 3))
  (def (scan1) (lightweight void void)
    (= this (move this))
    (scan0))
  (= (aref seed 0) 3)
  (= (aref seed 1) 4)
  (= (aref seed 2) 5)
  (for ((= i 0) (< i n) (inc i))
   (= k (nrand48 seed))
   (insert scan1 this k k)))
```

Figure 4.11: The LW-SC program for BinTree.

```
(deftype sht (ptr (lightweight void void)))
(def (bin2list scan0 x rest)
     (fn (ptr Alist) sht (ptr Bintree) (ptr Alist))
  (def a (ptr Alist) 0)
  (def kv (ptr KVpair) 0)
  (def (scan1) (lightweight void void)
    (= x (move x))
    (= rest (move rest))
    (= a (move a))
    (= kv (move kv))
    (scan0))
  (if (fref (mref x) right)
      (= rest (bin2list scan1 (fref (mref x) right)
                        rest)) )
  (= kv (getmem scan1 (ptr KVpair_d)))
  (= (fref (mref kv) key) (fref (mref x) key))
  (= (fref (mref kv) val) (fref (mref x) val))
  (= a (getmem scan1 (ptr Alist_d)))
  (= (fref (mref a) kv) kv)
  (= (fref (mref a) cdr) rest)
  (= rest a)
  (if (fref (mref x) left)
      (= rest (bin2list scan1 (fref (mref x) left)
                        rest)))
  (return rest) )
```

Figure 4.12: The LW-SC program for Bin2List.

```
(def (cpfib save0 n)
     (fn int (ptr (lightweight void)) int)
  (def pc int 0)
  (def s int 0)
  (def (save1) (lightweight void)
    (save0)
    (save-pc pc)
    (save-int s)
    (save-int n))
  (if (<= n 2)
      (return 1)
      (begin
       (= pc 1)
       (+= s (cpfib save1 (- n 1)))
       (= pc 2)
       (+= s (cpfib save1 (- n 2)))
       (return s))) )
```

Figure 4.13: The LW-SC program for fib(34).

Table 4.1: Performance measurements (for the creation and maintenance cost).

S:SPARC P:Pentium		Elapsed Time in seconds (relative time to plain C)				
		C	GCC	LW-SC	XCC	CL-SC
BinTree copying GC	S	0.180	0.263	0.192	0.181	0.249
		(1.00)	(1.46)	(1.07)	(1.00)	(1.38)
	P	0.152	0.169	0.156	0.150	0.179
		(1.00)	(1.11)	(1.03)	(0.988)	(1.18)
Bin2List copying GC	S	0.292	0.326	0.303	0.289	0.318
		(1.00)	(1.12)	(1.04)	(0.99)	(1.09)
	P	0.144	0.145	0.151	0.146	0.154
		(1.00)	(1.01)	(1.05)	(1.01)	(1.07)
fib(34) check-pointing	S	0.220	0.795	0.300	0.226	0.361
		(1.00)	(3.61)	(1.36)	(1.03)	(1.64)
	P	0.0628	0.152	0.138	0.0751	0.162
		(1.00)	(2.42)	(2.20)	(1.20)	(2.58)
nqueens(13) load balancing	S	0.478	1.04	0.650	0.570	1.05
		(1.00)	(2.18)	(1.36)	(1.19)	(2.20)
	P	0.319	0.428	0.486	0.472	0.544
		(1.00)	(1.34)	(1.52)	(1.48)	(1.71)

Bin2List (copying GC) converts a binary tree with 500,000 nodes into a linear list, with a copying-collected heap (Figure 4.12).

fib(34) (check-pointing) calculates the 34th Fibonacci number recursively, with a capability of capturing a stack state for check-pointing (Figure 4.13).

nqueens(13) (load balancing) solves the N-queens problem (N=13) on a load-balancing framework based on lazy partitioning of sequential programs [56, 57].

Note that nested functions are never invoked, that is, garbage collection, check-pointing and task creation do not occur, in these measurements because we measured the costs of creating and maintaining nested functions.

We measured the performance on 1.05 GHz UltraSPARC-III and 3GHz Pentium 4 using GCC with `-O2` optimizers. Table 4.1 summarizes the results of performance measurements, where "C" means the plain C program without high-level services, and "GCC" means the use of GCC's nested functions. The "XCC" means the use of XC-cube, which is an extended C language with some primitives added for safe and efficient shared memory programming [58]. XC-cube also features nested functions with lightweight closures [56, 57], which are implemented at the assembly language level by modifying GCC directly.[4] The "CL-SC" (closure SC) means the use of nested functions with *non-lightweight* closures, whose implementation is almost the same as LW-SC except that all local variables and parameters are stored into the explicit stack.

Since nested functions are created frequently in fib(34), LW-SC shows good performance on SPARC, compared to GCC where the cost of flushing instruction caches is significant. On the other hand, LW-SC shows not so good performance on Pentium 4 where overhead with additional operations in LW-SC is emphasized.

Since several local variables can get callee-save registers in BinTree, LW-SC shows good performance on SPARC, even if function calls (i.e. creations) are infrequent. This effect is not so significant in fib(34) since there is few local variable accesses in the `fib` function.

[4] The detail of its implementation is reported by a separate paper [59].

Table 4.2: Performance measurements (for the invocation cost).

		Elapsed Time in seconds				
		C	GCC	LW-SC	XCC	CL-SC
QSort (200,000)	SPARC	0.795	0.821	7.04	8.03	0.931
	(Ratio to C)	(1.00)	(1.03)	(8.86)	(10.1)	(1.17)
	Pentium	0.139	3.44	3.77	3.38	0.186
	(Ratio to C)	(1.00)	(24.7)	(27.1)	(24.3)	(1.33)
Bin2List copying GC	SPARC	—	0.495	0.522	0.495	0.526
	(GC time)		0.278	0.296	0.279	0.302
	Pentium	—	0.248	0.257	0.249	0.259
	(GC time)		0.0647	0.0685	0.0669	0.0714

LW-SC does not show good performance in nqueens(13) since unimportant variables are allocated to registers. Since Pentium 4 has only a few callee-save registers and performs explicit save/restore of callee-save registers which is implicit with SPARC's register window, the penalty of wrong allocation is serious.

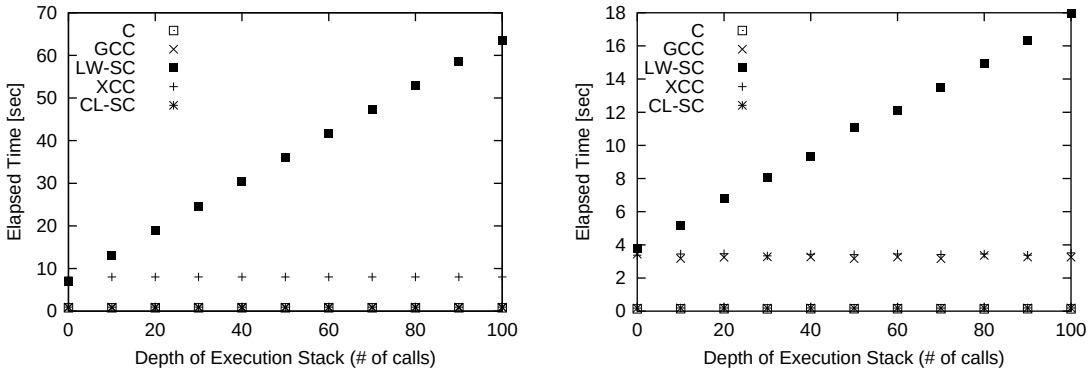
XC-cube shows better performance than LW-SC mainly because it does not employ some of additional operations in LW-SC, for example checking flags after returning from ordinary functions and at the beginning of function bodies (by using assembly-level techniques such as modifying return addresses). However, the difference is negligibly small if the body of a function is sufficiently large.

CL-SC shows worse performance than LW-SC since all local variables and parameters are stored in the explicit stack and they never get registers.

4.5.2 Invocation cost

To measure the cost of invoking nested functions, we employ the following programs:

QSort sorts 200,000 integers by the quick sort algorithm invoking a nested function as a comparator, whose owner is the caller of the sorting function (Figure 4.15). In the plain C program, the comparison function is defined as the ordinary function where `d` is declared as a global variable.



UltraSPARC-III Pentium 4

Figure 4.14: Elapsed time in QSort against the number of intermediate function calls.

```
(def (mod-sort a n d)
     (fn void (ptr int) int int)
  (def (comp-mod pp pq)
       (lightweight int (ptr void) (ptr void))
    (return
     (if-exp (< (% (mref (cast (ptr int) pp)) d)
                (% (mref (cast (ptr int) pq)) d))
             1
         (if-exp (== (% (mref (cast (ptr int) pp)) d)
                     (% (mref (cast (ptr int) pq)) d))
                 0
                 -1))))
  (quicksort a n (sizeof int) comp-mod))
```

Figure 4.15: The LW-SC program of QSort (calling the sorting function by passing a nested function `comp-mod` as a comparator).

Bin2List (copying GC) works as the same as Bin2List in Section 4.5.1, except that the garbage collector actually runs and nested functions are called for scanning the stack (therefore there is no plain C program). The collectors employ a simple breadth-first (non-recursive) copying GC algorithm.

Table 4.2 summarizes the results of performance measurements. In LW-SC, the invocation cost is high because saving (restoring) the values in the execution stack are necessary upon calling (returning from) nested functions, which causes bad performance in QSort. What is worse is that the cost of invoking a nested function increases depending on the depth of the execution stack at the time of the invocation. To show it clearly, we invoked `mod-sort` in Figure 4.15 on top of various numbers of intermediating

function calls (Figure 4.14). The result shows the elapsed time increases proportionally to the stack depth only in LW-SC. We think that the cost of throwing an exception to an exception handler may also change with a similar reason.

CL-SC shows good performance in QSort because the unwinding and the reconstructing the execution stack are unnecessary.

Notice that GCC on Pentium shows bad performance in QSort. We guess that this is because trampoline code placed in a writable data area (not a read-only program area) prevents the processor from prefetching instructions.

All implementations show almost the same performances in Bin2List even when only GC times are compared. This is because the invocation costs are negligible relative to the other costs for GC (such as scanning heaps).

These results show that LW-SC works effectively if nested functions are not so frequently called, and that CL-SC works better if they are called very often. Programmers and compiler writers can choose one of these implementations depending on their situation.

4.6 Implementation of High-level Services

This section shows language extensions which we implemented by utilizing LW-SC as an intermediate language. Chapter 6 shows a more significant example where dynamic load balancing is realized by our novel strategy.

4.6.1 HSC—Copying GC

To implement garbage collection, the collector needs to be able to find all *roots*, each of which holds a reference to an object in the garbage-collected heap. In C, a caller's pointer variable may hold an object reference, but it may be sleeping in the execution stack until the return to the caller. Even when using direct stack manipulation, it is difficult for the collector to distinguish *roots* from other elements in the stack.

By embedding garbage collection, we actually implemented *HSC* (High-level SC), which is a memory safe SC-1 language with objects allocated in a garbage collected heap.

Specification

To guarantee memory safety, we modified the specification of SC-1. In particular, HSC uses "references" instead of pointers. Therefore,

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

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

- The syntax `(new` *expression*`)` is added to *expression*, which evaluates a given expression, allocate an object initialized to the result, and then returns a reference to the object.
- An array type is not equivalent to any pointer types even if it appears as an argument type.
- An array reference
  `(aref` $expression_1$ $expression_2$`)`
  is permitted only if the value of $expression_1$ has an array type. It is no longer equivalent to
  `(mref (+` $expression_1$ $expression_2$`))`.
- No `union` types exist. (From a practical viewpoint, disjoint unions should be implemented instead.)

Figure 4.16 shows an example of an HSC program. In this figure, `(init Pair (struct e1 e2))` is an SC-0 expression which is equivalent to `(Pair){e1,e2}` in C99 [21].

Implementation

Figure 4.17 partially shows how scanning of *roots* can be implemented by using nested functions. `Getmem` allocates a new object in the heap and may invoke the copying collector

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

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

Figure 4.16: An HSC program.

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

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

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

Figure 4.17: Scanning stack implemented by nested functions in LW-SC.

with the nested function `scan1`. The copying collector can indirectly call `scan1`, which effects the evacuation (copying) of objects by using roots (`e1`, `e2` and `pair`) and indirectly calls `scan0` in a nested manner. The actual entity of `scan0` may be another instance of `scan1` in the caller. The nested calls are performed until the bottom of the stack is reached.

4.6.2 MT-SC—Multi-threading

We implemented an extended SC-1 language *MT-SC* involving features for multi-threading.

We used the implementation techniques which we proposed before in [45, 17].

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

Figure 4.18: An MT-SC program.

Specification

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

MT-SC has following primitives:

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

Figure 4.18 shows an example of an MT-SC program.

Implementation

These features can be implemented using nested functions in LW-SC as shown in Figure 4.19. Every function has its own nested function to continue its equivalent computation and

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

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

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

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

Figure 4.19: Multi-threading implemented by LW-SC.

saves the pointer of the nested function to be called later to early execute the thread's unprocessed computation (continuation). Such a nested function is also generated for each `thread-create`.

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

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

4.7 Related Work

4.7.1 Compiler-based implementations of nested functions

As described above, GCC also features nested function but it is less portable and takes high maintenance/creation costs. XC-cube implements nested functions with lightweight closures by modifying the GCC compiler. It shows better performance, but it also lacks portability.

4.7.2 Closure objects in modern languages

Many modern languages such as Lisp and ML implement closures as first class objects. Those closure objects are valid after exit of their owner blocks. In most implementations they require some runtime supports such as garbage collection, which makes it C too inefficient to be used as an intermediate language to implement high-level languages.

4.7.3 Portable assembly languages

C-- [23, 35] also has an ability to access the variables sleeping in the execution stack by using the C-- runtime interface to perform "stack walk." We expect that its efficiency is better than LW-SC, and almost equal to XC-cube. In terms of portability, LW-SC has an advantage that we can use pre-existing C compilers.

4.7.4 High-level services

This section lists high-level services which are important applications of nested functions and their implementation techniques in our work.

Garbage collection

As mentioned above, it is difficult to manipulate object references sleeping in the execution stack for garbage collection in C.

For this reason, conservative collectors [2] are usually used. Conservative copying collectors can inspect the execution stack but cannot modify it. Accurate copying GC can be performed by using translation techniques based on "structure and pointer" [14, 16], but it takes higher maintenance costs.

Capturing/Restoring stack state

Porch [43] is a translator that transforms C programs into C programs supporting portable checkpoints. Portable checkpoints capture the state of a computation in a machine-independent format that allows the transfer of computations across binary incompatible machines. They introduce source-to-source compilation techniques for generating code to save and recover from such portable checkpoints automatically. To save the stack state, the program repeatedly returns and legitimately saves the parameters/local variables until the bottom of the stack is reached. During restoring, this process is reversed. Similar techniques can be used to implement migration and first-class continuations.

As shown in Figure 4.13, the stack state can be captured without returning to the callers using nested functions. It uses similar techniques with the ones for scanning roots described above.

Multi-threads: latency hiding

Concert [33], OPA [49] use similar translation techniques to support suspension and resumption of multiple threads on a single processor with a single execution stack (e.g., for latency hiding). They create a new child thread as an ordinary function call and if

the child thread completes its execution without being blocked, the child thread simply returns the control to the parent thread. But in case of the suspension of the child thread, the C functions for the child thread legitimately saves its (live) parameters/local variables into heap-allocated frames and simply returns the control to the parent thread. When a suspended thread become runnable, it may legitimately restore necessary values from the heap-allocated frames.

The library implementation of StackThreads [48] provides special two service routines: `switch_to_parent` to save the context (state) of the child thread and transfer the control to the parent thread, and `restart_thread` to restore the context and transfer the control to the restarted thread. These routines are implemented in assembly languages by paying special attention to the treatment of callee-save registers.

StackThreads/MP [47] allows the frame pointer to walk the execution stack independently of the stack pointer. When the child thread is blocked, it can transfer the control to an arbitrary ancestor thread without copying the stack frames to heap. StackThreads/MP employs the unmodified GNU C compiler and implements non-standard control-flows by a combination of an assembly language postprocessor and runtime libraries.

Lazy Threads [11] employ a similar but different approach to frame management and thread suspension. Frames are allocated in "stacklet," which is a small stack for several frames. A blocked child thread returns the control to the parent without copying the stack frame to heap. When the parent is not at top of the stacklet, it first allocates a new stacklet for allocating a stack frame. Lazy Threads are implemented by modifying the GNU C compiler.

Load balancing

To realize efficient dynamic load balancing by transferring tasks among computing resources in fine-grained parallel computing such as search problems, load balancing schemes which lazily create and extract a task by splitting the present running task, such as *Lazy Task Creation* (LTC) [30], are effective. In LTC, a newly created thread is directly and immediately executed like a usual call while (the *continuation* of) the oldest thread in the computing resource may be stolen by other idle computing resources. Usually, the

idle computing resource (*thief*) randomly selects another computing resource (*victim*) for stealing a task.

Compilers (translators) for multi-threaded languages generate low-level code. In the original LTC [30], assembly code is generated to directly manipulate the execution stack. Both translators for Cilk [10] and OPA [49] generate C code. Since it is illegal and not portable for C code to directly access the execution stack, the Cilk and OPA translators generate two versions (fast/slow) of code; the fast version code saves values of live variables in a heap-allocated frame upon call (in the case of Cilk) or return (in the case of OPA) so that the slow version code can continue the rest of computation based on the heap-allocated saved *continuation*.

A message passing implementation [7] of LTC employs a polling method where the *victim* detects a task request sent by the *thief* and returns a new task created by splitting the present running task. This technique enables OPA [49], StackThreads/MP [47] and Lazy Threads [11] to support load balancing.

We restructure LTC with backtracking, where callers' variables are accessed by using nested functions for infrequent task creation [56, 57]. See Chapter 6 for more detail.

Chapter 5

Using Existing C Header Files in SC Based on Translation from C to SC

This chapter introduces *C2SC Compiler*, a translator from C to SC-1. This translator allows SC programmers to include existing C header files (`.h` files) as translated SC header files by using the `%cinclude` directive which is provided by SC preprocessors (see Section 2.3).

We first discuss the necessity of the translator in the next section, and then detail its implementation.

5.1 Why We Need Translation from C to SC

As we mentioned in Section 2.1, SC languages are designed not only as data structures of ASTs but also for human programming. Because SC-1 has the same semantics as C, existing functions defined in C should be able to be used in SC-1. But in order to use a C function, its declaration needs to be written in SC-1. Thus a "reverse" translator is required.

Note that the reverse translator is not required in the case where we just write a program in SC-1 (or SC-0), because the SC compiler (and also rule-set for SC-1-to-SC-0 transformation) does not perform any semantic analysis such as type checking. For example, the SC-1 statement multiplies a pointer and a floating point number, which causes a data type mismatch error:

```
(begin (def x double 3.14159)
       (def p (ptr int))
       (return (* p (sin x))))
```

But the SC compiler translates the code into C without any type checking:

```
{ double x=3.14159;
  int *p;
  return p * sin(x); }
```

This strategy is reasonable because we can resign such type checking to the back-end C compiler. All the SC compiler needs to do is add

```
#include <math.h>
```

at the head of the output C code. But this strategy is not sufficient when language extension is applied. For example, the `temp` rule-set, used to implement LW-SC in Section 4.4.2, translates

```
(= y (f (sin x) (cos x)))
```

into

```
(= tmp1 (sin x))
(= tmp2 (cos x))
(= y (f tmp1 tmp2)),
```

and the additional variable definitions of `tmp1` and `tmp2` need to be inserted before these expressions. The definitions require the types of these variables, that is, the return types of `sin` and `cos` which can be obtained only by looking up the header file `math.h`.

Another purpose of C2SC Compiler is to allow us to use macros defined by `#define` directives in C header files, which include object-like macros such as `NULL`, `stdin` and `stdout` and function-like macros such as `putchar` and `getchar`.

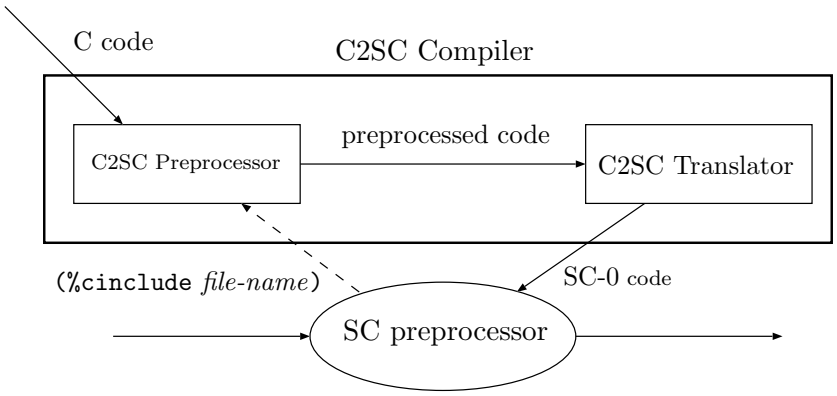


Figure 5.1: Translation flow in C2SC Compiler.

5.2 Implementation

5.2.1 Overview

Figure 5.1 shows the translation flow in C2SC Compiler. The SC preprocessor in this figure corresponds to the one in Figure 2.1. The `%cinclude` directive tells the SC preprocessor to call C2SC Compiler with a given file name as an input. Then C2SC Compiler translates C code in the file into SC-1 code and the result replaces the `%cinclude` expression.

The translation of C2SC Compiler consists of two phases; the first phase is performed by *C2SC Preprocessor* and the second phase is by *C2SC Translator*.

C2SC Preprocessor

C2SC Preprocessor, which we implemented by modifying an existing C preprocessor (MCPP [29]), translates input C code as follows.

Step 1 applies the translation corresponding to a normal C preprocessor, which includes

- including files specified by `#include` directives,
- expanding macros defined by `#define` directives and predefined macros, and

- invalidation of code fragments delimited by `#ifdef` and `#ifndef` and `#if`,

Step 2 translates each `#define` and `#undef` into an intermediate expression in the following manner:
`#define` *identifier repl-list*
⟶ `(%defconstant-cexp` *identifier string*`)`
`#define` *identifier*`(`*identifier-list*`)` *repl-list*
⟶ `(%defmacro-cexp` *identifier* `(`*identifier-list*`)` *string*`)`
`#undef` *identifier* ⟶ `(%undef` *identifier*`)`.
The result is saved separately from the result of Step 1. Here, *string* is a Lisp string corresponding to the code fragment, which is obtained by macro-expanding *repl-list* in the macro definition environment after the whole header file is processed.

The results of Step 2, the translated macro definitions, are inserted after Step 1 in order to prevent the SC preprocessor, which is called after C2SC Compiler, from re-macroexpanding the translated code.

Macro-expanding *repl-list* into *string* does not comply the standard specification of the C preprocessor [20]. Its necessity and drawbacks are discussed in Section 5.3.2.

C2SC Translator

Following steps 1 and 2 of C2SC Preprocessor, C2SC Translator, which is implemented in Common Lisp, translates the result of Step 1 and that of Step 2 into SC-1 code and SC macro definitions, respectively, and output them in this order.

The result of Step 1, the preprocessed C code, is lexed, parsed and translated into equivalent SC-1 code.

Each `%defconstant-cexp`/`%defmacro-cexp`, generated in Step 2, is translated into a `%defmacro`/`%defconstant` directive for the SC preprocessor. The C code fragment *string* in the `%defconstant-cexp`/`%defmacro-cexp`, is translated into an equivalent SC code fragment *if possible*, and is used as the body of the SC macro definition. If the translation is impossible, `%defconstant-cexp`/`%defmacro-cexp` is translated into

[Example 1]
```
#define apply_f(a) f(a)
```
[Example 2]
```
#define BEGIN {
#define END }
#define NOTOKEN
```
[Example 3]
```
#define concat_token(a,b) a ## b
```
[Example 4]
```
#define tp_cast(a) ((tp)(a))
```

Figure 5.2: C macros which are difficult to translate into SC macros.

`(%defmacro` *identifier* `(`*identifier-list*`) ~(c-exp` *string* `,@`*identifier-list*`))` or `(%defconstant` *identifier* `(c-exp` *string*`))`.

`C-exp` is an additional primitive of SC-0/1. The SC compiler translates a `c-exp` expression with a single argument into *string* (a C code fragment) as it is. If the `c-exp` expression has two or more arguments, the compiler replaces C identifiers in *string* which corresponds to the macro parameters (*identifier-list*) with the second and the following arguments of the `c-exp`. The translated macro can be used in SC, but cannot be analyzed nor transformed by SC translators.

C2SC Translator performs no modification for `%undef`s. They are evaluated as `%undef` directives by the SC preprocessor.

5.2.2 Translation of C macros into SC macros

Any preprocessed C code (result of Step 1 of C2SC Preprocessor) can be translated into SC-1 straightforwardly, because SC-1 and C have the same semantics.

However, translating `#define`s for the C preprocessor into `%defmacro`/`%defconstant` for the SC preprocessor is impossible in some cases. This is mainly because the *repl-list* in a `#define` is a token sequence, not a parsing unit. We cannot always construct a parse tree of the token sequence or, even if parsing is successful, the constructed parse tree may not be unique. Because the expansion for an SC macro must be written as an S-expression which roughly corresponds to a parse tree, a `#define` with such a token

sequence as the expansion cannot be translated into an SC macro. It is more difficult to translate a C macro which takes parameters because they are also replaced to any token sequences.

Figure 5.2 illustrates C macros which are difficult to translate into SC macros. Even simple macros such as Example 1 can be used in two ways:

`y = apply_f(x);` ⟶ `y = f(x);`

`long apply_f(int x);` ⟶ `long f(int x);`.

The corresponding SC-1 code fragments are `(= y (f x))` and `(decl (f x) (fn long int))` respectively. Thus, no SC macro corresponds to this C macro. Moreover, it is impossible to define an SC macro even only for the latter usage.

Example 2 shows C macros with token sequences of which we cannot construct parse trees: no parsing unit corresponds to a single brace, and also to an empty token sequence.

The macro in Example 3 uses a `##` operator, which concatenates two tokens into a single token. Such token concatenation cannot be translated to any operation over S-expressions.

The usage of the macro in Example 4 is not unique: it can be interpreted differently depending on the compile-time environment. An application of this macro is as follows:

`tp_cast(x);` → `((tp)(x));`.

The expanded code fragment is interpreted as a cast-expression to the type `tp` in an environment where `tp` is defined as a type name by `typedef` (`(cast tp x)` in SC-1). Otherwise, it is interpreted as a function call (e.g., `(tp x)` in SC-1). The SC preprocessor cannot determine which expansion is correct because it does not perform semantic analysis.

5.2.3 Countermeasures

As described above, translating C macro definitions into SC macro definitions is impossible in some cases. But from a practical viewpoint, the examples above are not fatal in most cases because of the following reasons:

- the macro in Example 1 is useless if this is defined for using in function declarations,
- the macros in Example 2 are also useless in S-expression based languages,

- token concatenation in Example 3 is commonly used for generating identifiers and it can be translated by limiting its usage to generating an identifier (Lisp symbol) from two atoms, and
- we seldom define a macro such as in Example 4 for the purpose of using the single macro for both cast-expressions and function calls.

In this work, we implemented C2SC Translator based on these insights. The translation is performed as follows:

Step 1 tries parsing a given C code fragment assuming it can be parsed as a sequence of declarations, a sequence of member declarations of a struct/union, a type name, a statement, or an expression (these candidates correspond to the nonterminating symbols of BNF used for the C language definition [20]: *translation-unit*, *struct-declaration-list*, *type-name*, *statement* and *expression*, respectively).

Step 2 gives up translation if the parsing fails for all the candidates. In this case, an SC macro definition with a `c-exp` is output.

Step 3 if the parsing succeeded for only one candidate, translates the parsing tree into an SC-1 code fragment.

Step 4 if the parsing succeeded for two or more candidates, prompts the user to choose one of the translated results and outputs the choice.

In Step 1, a macro parameter in the code fragment is regarded as an *identifier*. For example, in parsing the code fragment in the following macro definition,

```
#define X_OR_Y(x,y) ((x)||(y))
```

`x` and `y` are treated as *identifier*s. For this example, the parsing succeeds as an *expression* and the SC macro definition

```
(%defmacro X_OR_Y (x y) ~(or ,x ,y))
```

will be output as a result. But the definition

```
#define X_OP_Y(op) (x op y)
```

in which `op` is expected to be replaced by an operator, cannot be translated because `(x op y)` cannot be parsed as any of the parsing unit listed above since `op` is regarded as an identifier. The transformation result is

```
(%defmacro X_OP_Y (op) ~(c-exp "(x op y)" ,op)).
```

A `##` operator is translated into an application of the Lisp function which concatenates two S-expressions into a Lisp symbol. For example, the macro definition in Example 3 in Figure 5.2 will be translated into

`(%defmacro concat_token (a b) (concat-symbol a b))`,

where `concat-symbol` is the concatenation function. A `#` operator, which stringifies the token after it, is also translated into an application of the Lisp function that stringifies a given S-expression.

For a macro definition such as Example 4, the translator tries parsing for two cases for each appearing identifier: when the identifier is defined as a type name and when it is not defined. If the parsing succeeded for both cases, the translator adds the results to the list of candidates from which the user chooses in Step 4.

To avoid prompting, the following pragmas for C2SC Compiler can be used.

- `#pragma c2sc_typename` *identifier*
- `#pragma c2sc_not_typename` *identifier*
- `#pragma c2sc_query_typename` *identifier*

The *identifier* specified by `c2sc_typename` is assumed to be a type name in translating the macro definitions that follow. The *identifier* specified by `c2sc_not_typename` is assumed not to be a type name. The effects of these programs are canceled by the `c2sc_query_typename` pragma.

For example, in translating the macro definition in Example 4, the translator prompts the user with the following message to choose 1 or 2.

```
1: ~(cast tp ,a)
2: ~(tp ,a)
>>
```

This prompting can be avoided by putting

```
#pragma c2sc_typename tp
```

before the `#define` directive (1 is automatically chosen).

5.3 Evaluation and Discussion

5.3.1 Translation results from the standard POSIX header files

To evaluate practicality of C2SC Compiler, we applied it to the C header files defined in POSIX [19], listed in Table 5.1. We used the header files for GCC 3.4.2 on FreeBSD 5.3-STABLE. The header files which are not listed in Table 5.1 but included by nested `#include` directives are also applied. We removed the code fragments with GCC extensions such as `__attribute__` and `__extension__` from the header files. The verification whether the translation is correct is performed by hand.

The result is as follows. The compiler translated all the C code correctly, except for the macro definitions. Although most of the macro definitions are translated correctly, some definitions failed to translate and some caused prompting. These definitions are listed below.

- **The macros that failed to translate**

  A macro definition without tokens such as

  ```
  #define _STDIO_H
  ```

  cannot be translated in our strategy, but they would not cause fatal problems because such a macro is only used in predicates of `#ifdef` and `#ifndef` etc. Figure 5.3 shows the macros that cannot be translated for the other reasons.[1]

  The macros in group (1) failed to translate because each replacement consists only of an operator, and the macros in (2) failed because each expects its parameter

[1] The replacements shown in the macro definitions in Figure 5.3 and Figure 5.4 have been macro-expanded by C2SC Preprocessor.

Table 5.1: The header files used for evaluation.

<aio.h>	<arpa/inet.h>	<assert.h>
<complex.h>	<cpio.h>	<ctype.h>
<dirent.h>	<dlfcn.h>	<errno.h>
<fcntl.h>	<fenv.h>	<float.h>
<fmtmsg.h>	<fnmatch.h>	<ftw.h>
<glob.h>	<grp.h>	<iconv.h>
<inttypes.h>	<iso646.h>	<langinfo.h>
<libgen.h>	<limits.h>	<locale.h>
<math.h>	<monetary.h>	<mqueue.h>
<ndbm.h>	<net/if.h>	<netdb.h>
<netinet/in.h>	<netinet/tcp.h>	<nl_types.h>
<poll.h>	<pthread.h>	<pwd.h>
<regex.h>	<sched.h>	<search.h>
<semaphore.h>	<setjmp.h>	<signal.h>
<spawn.h>	<stdarg.h>	<stdbool.h>
<stddef.h>	<stdint.h>	<stdio.h>
<stdlib.h>	<string.h>	<strings.h>
<stropts.h>	<sys/ipc.h>	<sys/mman.h>
<sys/msg.h>	<sys/resource.h>	<sys/select.h>
<sys/sem.h>	<sys/shm.h>	<sys/socket.h>
<sys/stat.h>	<sys/statvfs.h>	<sys/time.h>
<sys/timeb.h>	<sys/times.h>	<sys/types.h>
<sys/uio.h>	<sys/un.h>	<sys/utsname.h>
<sys/wait.h>	<syslog.h>	<tar.h>
<termios.h>	<tgmath.h>	<time.h>
<trace.h>	<ucontext.h>	<ulimit.h>
<unistd.h>	<utime.h>	<utmpx.h>
<wchar.h>	<wctype.h>	<wordexp.h>

to be a string literal which is expected to be concatenated with the surrounding string literals to make a single string (the C preprocessor concatenates adjacent string literals into a single one).

The macros in (3) are expected to be used for initializers of structs, and are typically used in definitions like:

```
pthread_once_t once_control PTHREAD_ONCE_INIT;
```

This definition fails to be translated because it is not any of the parsing unit that are tried in Step 1 in Section 5.2.3.

The macro in (4), where `cmp` is expected to be an operator, is the same kind of macro as `X_OP_Y` in Figure 5.2. It cannot be translated because of the reason described above.

(1)
<iso646.h>
```
#define and    &&
#define and_eq &=
#define bitand &
#define bitor  |
#define compl  ~
#define not    !
#define not_eq !=
#define or     ||
#define or_eq  |=
#define xor    ^
#define xor_eq ^=
```

(2)
<sys/cdefs.h>
```
#define __COPYRIGHT(s) __asm__ (".ident\t\"" s "\"")
```
and 8 similar macros.

(3)
<pthread.h>
```
#define PTHREAD_ONCE_INIT {0, ((void *)0)}
```
2 similar macros in <netinet/in.h>,
6 similar macros in <netinet6/in6.h>, and
2 similar macros in <socket.h>.

(4)
<sys/time.h>
```
#define timercmp(tvp,uvp,cmp) \
  (((tvp) -> tv_sec == (uvp) -> tv_sec) ? \
   ((tvp) -> tv_usec cmp (uvp) -> tv_usec) : \
   ((tvp) -> tv_sec  cmp (uvp) -> tv_sec))
```

Figure 5.3: The macros that failed to be translated.

- **The macros that caused prompting**

  Totally 52 macros caused prompting while they are translated. Figure 5.4 shows only a few of them,[1] but they cover all the reasons of the prompting.

  The macros in (5) can be interpreted in two ways depending on whether an identifier is defined as a type name. For example, `isascii` can be interpreted in the following ways:

  ```
  ~(== (cast ,c (ptr (bit-not 0x7F))) 0)
  ~(== (bit-and ,c (bit-not 0x7F)) 0)
  ```

  However, it seems to be expected that `c` is not a type name and that this macro should be interpreted as the latter.

(5)

```
<sys/_sigset.h>
 #define _SIG_IDX(sig)  ((sig)-1)
 #define _SIG_WORD(sig) (((sig)-1)>>5)
 #define _SIG_BIT(sig)  (1<<(((sig)-1)&31))
<sys/signal.h>
 #define sigmask(m) (1<<((m)-1))
<sys/select.h>
 #define _howmany(x,y) (((x)+((y)-1))/(y))
<sys/types.h>
 #define minor(x) ((int)((x)&0xffff00ff))
<ctype.h>
 #define isascii(c) (((c)&~0x7F)==0)
 #define toascii(c) ((c)&0x7F)
```

(6)

```
<wctype.h>
 #define WEOF ((wint_t)-1)
```

(7)

```
<sys/cdefs.h>
 #define __offsetof(type,field) ((size_t)(&((type*)0)->field))
 #define __rangeof(type,start,end)  \
   (((size_t)(&((type*)0)->end)) -  \
    ((size_t)(&((type*)0)->start)))

<netinet/in.h>
 #define IN_CLASSA(i) (((u_int32_t)(i)&0x80000000)==0)
 #define IN_CLASSB(i) (((u_int32_t)(i)&0xc0000000)==0x80000000)
```

(8)

```
<stdio.h>
 #define feof(p)     \
  (!__isthreaded?    \
   (((p)->_flags&0x0020)!=0):(feof)(p))
 #define ferror(p)   \
  (!__isthreaded?    \
   (((p)->_flags&0x0040)!=0):(ferror)(p))
```

(9)

```
<sys/select.h>
 #define FD_ZERO(p) do {                          \
   fd_set *_p;                                    \
   __size_t _n;                                   \
   _p = (p);                                      \
   _n = (((1024U)+(((sizeof(__fd_mask)*8))-1)) \
         /((sizeof(__fd_mask)*8)))                \
   while (_n > 0)                                 \
     _p->__fds_bits[--_n] = 0;                    \
 } while (0)
```

Figure 5.4: The macros that caused prompting.

The `WEOF` macro in (6) can be interpreted in two ways:

```
~(cast wint_t (- 1))
~(- wint_t 1)
```

In this case, the former seems the right one since `wint_t` seems a type name.

The macros in (7) and (8) are the instances of Example 4 in Figure 5.2. For example, the `__offsetof` macro can be interpreted in two ways:

```
~(cast size_t (ptr (fref (mref (cast (ptr ,type) 0)) ,field)))
~(size_t (ptr (fref (mref (cast (ptr ,type) 0)) ,field)))
```

It seems that the macros in (7) are interpreted as cast expressions and that the macros in (8) as function calls.

In the `FD_ZERO` macro in (9), the code fragment `fd_set *_p;` causes prompting. This is interpreted as a variable declaration

```
(decl _p (ptr fd_set))
```

if `fd_set` is a type name, and otherwise interpreted as a multiplicative expression `(* fd_set _p)`.

The former seems to be correct.

All of these prompting can be avoided by pragmas described in Section 5.2.3. In addition, the translator can determine the interpretation for some of the macros in (1) by the knowledge of the C language, e.g., the `&` operator cannot be applied to a number literal.

As the result, C2SC Compiler is sufficiently practical; we can use all the C header files except the macros listed in Figure 5.3. The macros in (1), (3) and (4) in this figure also can be translated by changing the assumption, for example, by assuming a given token sequence may be translated as an operator. We can provide features to allow a programmer to modify the assumption by using annotation.

5.3.2 Safety of translation

Because the translation strategy described in the previous section is based on a certain set of assumptions, it is not sure that C macros are translated correctly. Note that an

SC program is translated correctly as long as incorrectly-translated macros are not used in the SC program. Using the macros in C does not cause any problems because they are expanded by C2SC Preprocessor.

Therefore, macros which C2SC Translator gave up to translate do not cause any problems other than that an SC programmer cannot use the macros, as long as the translator informs the programmer of the fact. These macros are translated into SC macros with `c-exps`, which the programmer can use if necessary. In particular, they are valuable for using in predicates of `%ifdef`, `%ifndef` and `%if` directives.

A problem arises when macros are translated differently from their original intentions. For example, the C macro

```
#define EQ_AB a=3, b=4
```

is translated into

```
(%defconstant EQ_AB (exps (= a 3) (= b 4))
```

by C2SC Translator, which seems to be correct. But this macro may be expected to be used for defining enumeration types like this:

```
enum eabc { EQ_AB, c }; ⟶ enum eabc { a=3, b=4, c };
```

The SC code fragment

```
(def (enum eabc) EQ_AB c)
```

is macro-expanded into

```
(def (enum eabc) (exps (= a 3) (= b 4)) c),
```

which is syntactically incorrect;

```
(def (enum eabc) (a 3) (b 4) c)
```

is correct.

Such incorrect translation occurs when parsing for a code C fragment succeeded accidentally even though the fragment is not any one of the candidates listed in Step 1 in Section 5.2.3; a sequence of declarations, a sequence of member declarations of a struct/union, a type name, a statement, and an expression. As long as the macro is intended to be used as any one of them, the macro can be used safely in SC.

However, the problem still remains in the case that the use of a macro causes nested macro expansion. For example, the C macro definition in (1) in Figure 5.5 is translated into the SC macro definition by C2SC Translator. When this macro is used, whether and

(1)
```
#define EXIT_IF_A(x)  \
  do {  \
    enum { EQ_AB, c } _t=(x);  \
    switch(_t) {  \
      case a:  exit(1); }  \
} while(0)
```

(2)
```
(%defmacro EXIT_IF_A (x)
  ~(do-while 0
    (def _t (enum EQ_AB c) ,x)
    (switch _t
      (case a)  (exit 1))))
```

(3)
```
#define EXIT_IF_A(x)  \
  do {  \
    enum { a=3, b=4, c } _t=(x);  \
    switch(_t) {  \
      case a:  exit(1); }  \
} while(0)
```

(4)
```
(%defmacro EXIT_IF_A (x)
  ~(do-while 0
    (def _t (enum (a 3) (b 4) c) ,x)
    (switch _t
      (case a)  (exit 1))))
```

Figure 5.5: Nested macro expansion.

how an expression such as `EQ_AB` is expanded depend on a macro definition environment on its expansion [20]. `EXIT_IF_A` cannot be used safely if `EQ_AB`, defined above, is defined as a C macro.

We solved this problem by making C2SC Preprocessor macro-expand the token sequence in a macro definition, as described in Section 5.2.1. Actually, the `EXIT_IF_A` macro definition is translated by the preprocessor into the definition written in (3), then translated by C2SC Translator into the SC macro definition written in (4), where the original intention is not changed.

The header files used in our evaluation in Section 5.3.1 actually includes the macro definitions

```
#define  _IOC(inout,group,num,len) \
 ((unsigned long)                  \
```

```
   (inout                          \
    | ((len & IOCPARM_MASK) << 16)  \
    | ((group) << 8) | (num)))
 #define  _IOW(g,n,t)              \
   _IOC(IOC_IN, (g), (n), sizeof(t))
 #define TIOCFLUSH _IOW('t', 16, int).
```

`TIOCFLUSH` can be translated correctly because `_IOW` and `_IOC` are macro-expanded in advance by C2SC Preprocessor. (Note that `_IOW(...)` cannot be parsed as a function call because its third argument is the type name `int`.)

This mechanism breaks the standard specification of the C preprocessor, as described above. But this is not fatal because it only causes a loss of ability to modify how nested expansion is performed when a macro defined in C is used, in an SC program.

Except for translation of macro definitions, C2SC Compiler can translates any C code correctly. Thus we can use type information and functions[2] defined in C header files safely in a SC program.

5.3.3 Prompting by multiple candidates

As described in Section 5.2.3, C2SC Translator prompts a user if there are multiple possible translations. To avoid such prompting, the translator can save all the candidates temporarily; and then the appropriate translation is chosen later automatically, depending on each compilation environment on expansion.

Such selection method must be defined in rule-sets for language extension, because the SC preprocessor cannot perform any semantic analysis of SC programs. Thus we must implement this mechanism carefully not to complicate the specification of rule-sets.

[2] Actually, a C header file often includes function definitions with the `inline` attribute.

5.4 Related Work

5.4.1 Foreign function interfaces

There exist many implementations which allow us to call C functions from another language. Such a mechanism is called *Foreign Function Interface (FFI)*. For example, GCC [41] allows us to call C functions from Fortran. GCC achieves this by generating an external name for each function by the specified naming rule. We can use foreign function call by defining and calling a function by names by the naming rule.

Some implementations of Common Lisp also provide FFI. CMU Common Lisp [27] is one of such implementations. The C function defined by

```
void foo( int x ){ ... }
```

can be called from Common Lisp by evaluating the following form:

```
(alien-funcall (extern-alien "foo" (function void int))
               10)
```

However, in the both cases, type information defined in header files cannot be used in the foreign languages because C functions are loaded by linking object files, which do not include such information any longer. Thus, a programmer need to write the type of a calling function (`(function void int)` in the above Lisp example) and (s)he is responsible for matching the number and the types of the function's parameters to the original definition.

Our translator takes in such type information automatically and outputs it as SC declarations. We can also output the information as code of another language so that a programmer of the language can call a C function without writing its type. It may be possible to use C macros from another language in the case that the foreign language is similar to C.

5.4.2 Including C header files in C++

We can include C header files in a C++ program using the `#include` directive for the C++ preprocessor. The C++ compiler can compile most of C programs correctly with

little modification; all we have to do is enclose the C code in `extern "C" {... }` to prevent external names from being modified.

This mechanism is similar to ours in that the both allow us to use C header files from another language.

Chapter 6

Backtracking-based Load Balancing

6.1 Introduction

For efficient parallel computing, computations should be allocated so that every computing resource—such as a core of a multi-core processor in a node of a cluster—has its own work at any time. But it is often difficult to predict such appropriate allocations statically especially in a heterogeneous environment or an environment where computing resources can join and leave dynamically. Irregular applications, such as tree-recursive algorithms and backtrack search algorithms, also make the prediction difficult.

Therefore dynamic load balancing is required, where a computation is dynamically allocated to idle computing resources. For efficient dynamic load balancing, the computation should be divided into larger tasks to reduce the total sum of dividing/allocation costs. It is also important to reduce management costs, which include costs of managing data that are necessary for spawning a task.

This paper proposes a scheme, called backtracking-based load balancing, which realizes these requirements. In our scheme, a worker basically performs a computation sequentially, but when it receives a task request from another idle worker, it creates a new task by dividing the remaining computation and returns the new task. In tree-recursive computation, the spawned task can be larger, in general, if we use information near the bottom of the execution stack. For this reason, the worker performs temporary backtracking before creating the task.

Multithreaded languages, such as Cilk [10] and MultiLisp [13], are widely accepted for parallel programming in shared memory environments. However, such languages have some overheads compared to sequential languages. In sequential languages, a single working space such as arrays is often reused for later computations; this common technique in sequential computing naturally improves spatial locality and achieves higher performance. But in multithreaded languages, at least, *logical* threads are created as potential tasks even when efficient LTC(Lazy Task Creation)[30]-based implementations are employed. Such a logical thread cannot *concurrently* use the working space for the parent thread. In our scheme, the worker basically does not create logical threads; rather it spawns an actual task (a piece of work for an idle worker) only when it receives a task request. On spawning a task, it performs necessary work-space allocations. In addition, our scheme does not need to manage any queue of logical threads. Therefore quite low overhead can be achieved.

We also propose a programming model in consideration of backtracking, which allows programmers to write undo-redo operations to be executed in backtracking. This allows fine-grained parallel programs for various backtrack search algorithms to be written elegantly and run very efficiently because they can not only reuse a single working space but also delay copying between working spaces by using backtracking.

In addition, we adopted message passing for communication among workers which exchange tasks and results as serialized task objects. This allows a single program to run in both shared and distributed (and also hybrid) memory environments with reasonable efficiency and scalability.

We designed and implemented such a language as an extended C language with new constructs.

6.2 Motivating Examples

We present two examples of tree-recursive fine-grained parallel computing; these applications are used to explain the details of our proposal in the following sections. They are also used for performance measurements. The first example is to recursively compute

```
int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   s1 = fib(n - 1);
   s2 = fib(n - 2);
   return s1 + s2;
 }
}
```

Figure 6.1: A C program for Fibonacci.

```
int a[12];   // manage unused pieces
int b[70];   // board
const int ps[57][5] = {...};
 // ps[i] represents the shape of the i-th (piece, direction)
const int pos[13] = {...};
 // ps[i] for pos[p]<=i<pos[p+1] corresponds the shape for the p-th piece

// Try from the j0-th piece to the 12-th piece in a[].
// The i-th piece for i<j0 is already used.
// b[k] is the first empty cell in the board.
int search (int k, int j0)
{
  int s=0; // the number of solutions
  for (int p=j0; p<12; p++) {  // iterate through unused pieces
    int ap=a[p];
    for (int i=pos[ap]; i<pos[ap+1] ; i++) {
      // examine the "i-th" (piece, direction) at the first empty location "k"
      ... local variable definitions ...
      if (Can ps[i] be set into the board b?);
      else continue;
      Set the piece ps[i] into the board b and update a.
      kk = the next empty cell;
      if (no empty cell?) s++;          // a solution found
      else s += search (kk, j0+1);    // proceed to the next piece
      Remove the piece from b and restore a (backtracking).
    }
  }
  return s;
}
```

Figure 6.2: A C program for Pentomino.

the n-th term of the Fibonacci sequence defined as follows:

$$\text{fib}(1) = \text{fib}(2) = 1$$
$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \text{ for } n > 2$$

Figure 6.1 shows a sequential C program for the n-th Fibonacci number. In each function call, computations of $\text{fib}(n-1)$ and $\text{fib}(n-2)$ can be executed in parallel.

The second example is a search algorithm to find all possible solutions to the Pentomino puzzle. A pentomino consists of five squares attached edge-to-edge. There are

```
// The structure of task objects.
struct tfib {
  int n;  // input
  int r;  // output
};
// The entry point of a task.
void exec_fib_task (struct tfib *pthis)
{ pthis->r = fib (pthis->n); }

int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   if (choose not to spawn?) {
     s1 = fib(n - 1);
     s2 = fib(n - 2);
   } else {
     Allocate a working space of struct tfib as this.
     this.n = n - 2;  // put the input value
     Send this as a newly spawned task.
     s1 = fib(n - 1);
     Wait and receive the result of this.
     s2 = this.r;      // get the output value
     Deallocate this working space.
   }
   return s1 + s2;
 }
}
```

Figure 6.3: A naively-parallelized program for Fibonacci.

twelve pentominos of different shapes. The Pentomino puzzle is to fill the 6×10 rectangular board with the twelve pentominos. This problem represents many similar search problems. Figure 6.2 shows a sequential C program for this problem. Each function call iterates through unused pieces (the outermost loop) and their directions (the inner loop). Parallelization seems applicable to the outermost loop. But there is an important difference from Fibonacci that this program does backtrack search where states of the board and the pieces are stored in working spaces: a piece is set at the next available position by one-step extension and removed by backtracking.

A naive task-parallel program for Fibonacci can be written as in Figure 6.3. In fib(n), each worker *chooses* whether it executes `fib(n-2)` by itself or it spawns a `fib(n-2)` task. For efficient load balancing, each task should be as large as possible so that a *minimum sufficient* number of tasks are created to keep all workers busy during the entire running time. That is, for each task, its worker should choose to spawn a *proper* number of tasks in the early stage and then choose not to spawn any more tasks except for adjusting the completion time. Such a strategy is infeasible without using precise information on the

whole execution. Thus, this naive approach does not work.

Figure 6.4 shows a naive task-parallel program for Pentomino. For parallelization, the outer `for` loop in Figure 6.2 is replaced with a `PAR_LOOP` macro. In `PAR_LOOP`, each worker *chooses* whether it performs all iterations or it spawns a task for the upper half of iterations. (In the latter case, it has another choice on the remaining lower half of iterations with the `PAR_LOOP` macro recursively.) This naive approach does not work since it requires an infeasible strategy as well.

In the following sections, we propose our approach with a feasible strategy for efficient load balancing. Note that the worker that executes a spawned task often requires its own initialized (copied) working space as in Figure 6.4. This motivated us to make an additional innovation of our approach.

6.3 Our approach

We propose a programming and execution framework called "Tascell." Tascell stands for *task cell*, which indicates that running tasks are divided like biological cells. In Tascell, we can spawn a task *lazily* by using *backtracking*.

The sequential computation of the C program in Figure 6.1 (Figure 6.2) is illustrated as a depth-first, left-to-right traversal of the *invocation tree* in the upper part of Figure 6.5 (Figure 6.6). Notice that the naive parallel program in Figure 6.3 (Figure 6.4) involves the same traversal if its worker always chooses not to spawn a task.

In Tascell, the worker always chooses not to spawn *at first*, but when it receives a task request, it spawns a task as if *it changed the past choice*. That is, as is shown in Figure 6.5,

1. it *backtracks* (goes back to the past),
2. it spawns a task (and changes the execution path to receive the result of the task),
3. it returns from the backtracking (restores the time), and
4. then it restarts its own task.

```
// The structure of task objects.
// Each worker has to have its own board for parallelization.
struct pentomino {
  int s;     // output
  int k, i0, i1, i2;
  int a[12]; // manage unused pieces
  int b[70]; // board
};

exec_pentomino_task (struct pentomino *pthis)
{
   pthis->s = search(pthis->k, pthis->i0, pthis->i1, pthis->i2, pthis);
}

const int ps[57][5] = {...};  // see Figure 6.2
const int pos[13] = {...};    // see Figure 6.2

#define PAR_LOOP (_i1, _i2, _body) {
   if (choose not to spawn?) {
     for(; _i1 < _i2; _i1++) _body
   } else {
     int _ih = (_i1 + _i2) / 2;
     int i1 = _ih;  // range for the new sub-task i1--i2
     int i2 = _i2;
     Allocate a working space of struct pentomino as this.
     { // put task inputs for upper half iterations
       copy_piece_info (this.a, tsk->a); // copy the
       copy_board (this.b, tsk->b);      // working space
       this.k = k; this.i0 = j0;
       this.i1 = i1; this.i2 = i2;
     }
     Send this as a newly spawned task.
     // lower half iterations (expanded n times for n-bit int)
     PAR_LOOP (_i1, _ih, _body)
     Wait and receive the result of this.
     s += this.s;   // get the result
     Deallocate this working space.
   }
}
// Try from the j1-th piece to the j2-th piece in a[].
// The i-th piece for i<j0 is already used.
// b[k] is the first empty cell in the board.
int search (int k, int j0, int j1, int j2, struct pentomino *tsk)
{
  int s=0; // the number of solutions
  int p=j1;
  PAR_LOOP(p, j2, {
    int ap=tsk->a[p];
    for (int i=pos[ap]; i<pos[ap+1] ; i++) {
      // examine the "i-th" (piece, direction)
      ... local variable definitions ...
      if (Can ps[i] be set into the board tsk->b?);
        else continue;
      Set the piece ps[i] into the board tsk->b and update tsk->a.
      kk = the next empty cell;
      if (no empty cell?)  s++;            // a solution found
      else                                 // the next piece
        s += search (kk, j0+1, j0+1, 12, tsk);
      Remove the piece from tsk->b and restore tsk->a (backtracking).
    }
  })
  return s;
}
```

Figure 6.4: A naively-parallelized program for Pentomino.

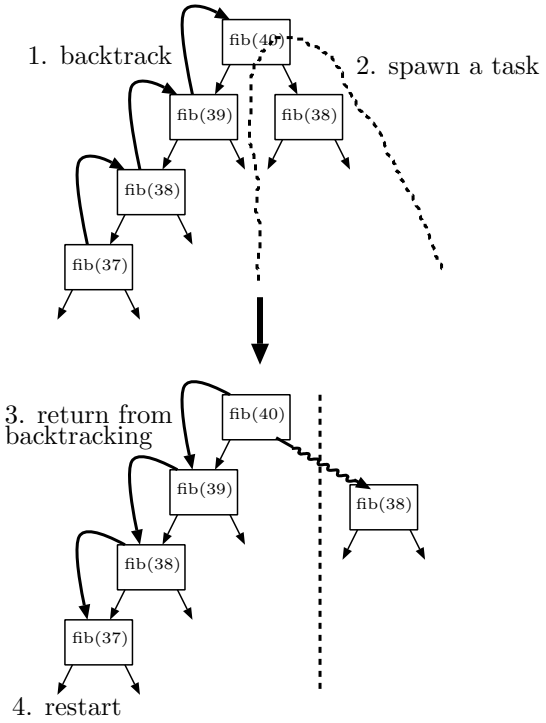


Figure 6.5: An task partitioning of a computation of fib(40) in Tascell; a new task for a computation of fib(38) is spawned.

Notice that we can spawn a larger task (as is the fib(38) subtree in the lower right part of Figure 6.5), in general, by backtracking to the oldest available choice point.

A Pentomino worker performs a sequential computation efficiently with its own working space by setting a piece and by removing the piece (i.e., backtracking or *undo*ing) across search steps. When the worker spawn a task, it must copy (part of) the "current" contents of its working space into a newly allocated space for the new task as in Figure 6.4. In our approach, the "current" contents should be equal to the past contents at the time of the past choice. As is shown in Figure 6.6, the worker can recover the past contents by performing proper *undo* operations along with backtracking as part of Step 1, it spawns a task with a copy of its working space at Step 2, and then it performs proper *redo* operations as part of Step 3 to restart its own task at Step 4.

Multithreaded languages such as Cilk [10] and MultiLisp [13] uses the technique called

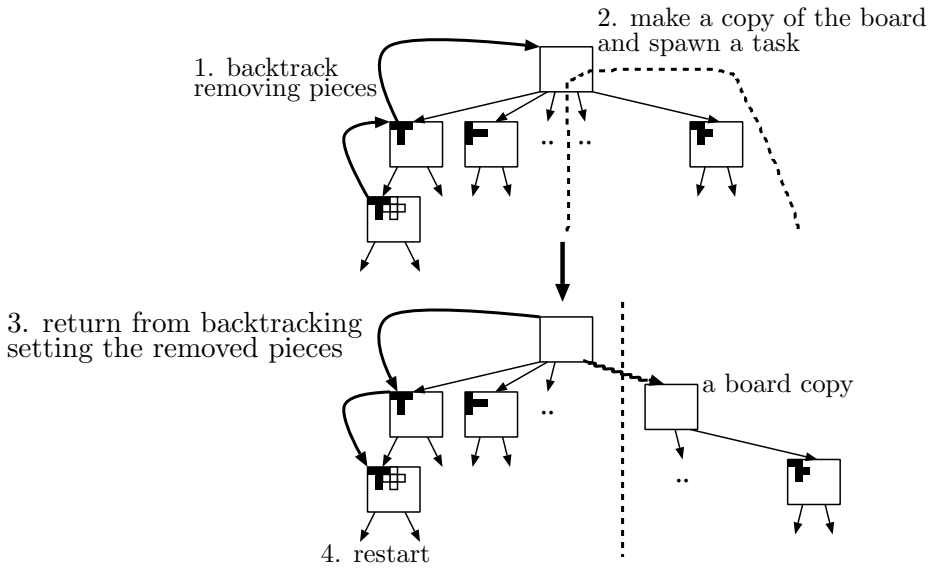


Figure 6.6: An task partitioning of Pentomino in Tascell; a new task for half iterations in the first step is spawned.

Lazy Task Creation (LTC) [30] to reduce overheads for dynamic load balancing. Our approach differs from LTC in the following points:

- Our worker performs a sequential computation unless it receives a task request. Because no *logical threads* are created as potential tasks, the cost of managing a queue for them can be eliminated.
- In multithreaded languages, each (logical) thread requires its own working space. In contrast, our worker can reuse a single working space while it performs a sequential computation to improve spatial locality and achieve higher performance.
- When we implement a backtrack search algorithm in multithreaded languages, each thread often needs each its own copy of its parent thread's working space. In contrast, our worker can delay copying between working spaces by using backtracking.
- Our approach supports (heterogeneous) distributed memory environments without

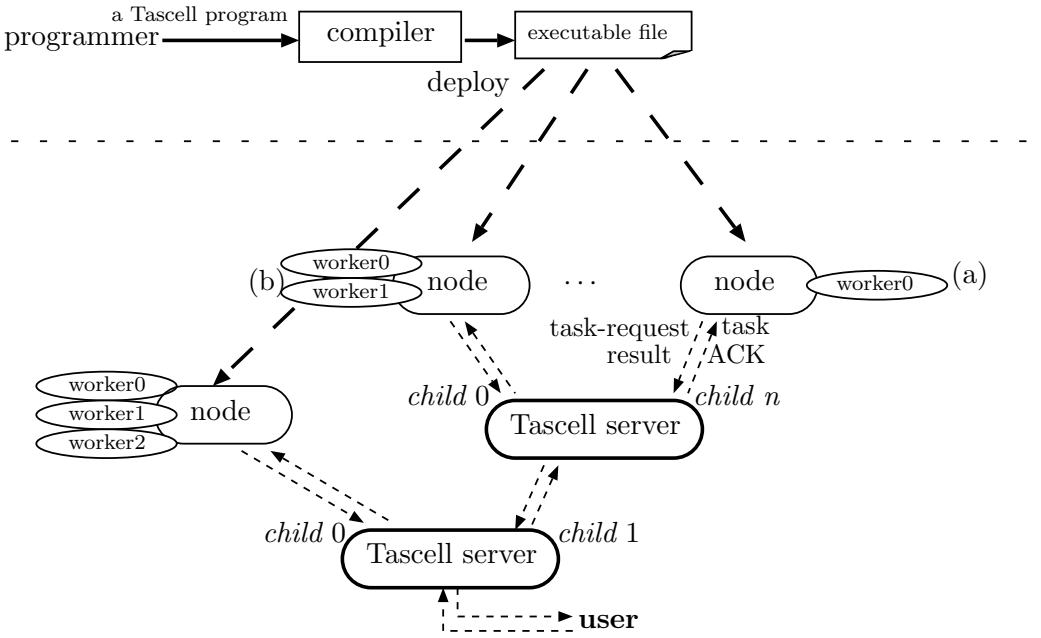


Figure 6.7: A multi-stage overview of the Tascell framework.

using distributed shared memory systems.

6.4 Tascell Framework

We implemented the Tascell framework to realize our idea, which consists of the Tascell server and the compiler for the Tascell language.

6.4.1 Overview

Figure 6.7 shows a multi-stage overview of the Tascell framework. Compiled Tascell programs are executed on one or more computation nodes. Each computation node has one or more worker(s) in the shared memory environment (the number can be specified as a runtime option).

Workers can communicate with each other by message passing. For automatic load

```
// The definition of a task in Tascell
task tfib {
  in:  int n;  // input
  out: int r;  // output
};
// The entry point of tfib.
// The task object this is declared implicitly.
task_exec tfib
{ this.r = fib (this.n); }

worker int fib (int n) {
 if (n <= 2) return 1;
 {
   int s1, s2;
   do_two          // construct in Tascell
     s1 = fib(n - 1);
     s2 = fib(n - 2);
   tfib {  // The task object this is declared implicitly.
     // put part (performed before sending a task)
     { this.n = n - 2; }
     // get part (performed after receiving the result)
     { s2 = this.r; }
   }                 // end of do_two
   return s1 + s2;
 }
}
```

Figure 6.8: A Tascell program for Fibonacci.

balancing, idle workers request tasks of busy workers. To relay a task request message for another node, Tascell servers which received the task request guess a busy node and pass the request to it, maybe via another server. Note that such a series of messages are exchanged automatically; programmers need not (and cannot) treat each of them directly.

Each task or its result is transmitted as a task object whose structure is defined in a Tascell program. If a request is from the same node, the object can be passed quickly via shared memory, otherwise it is transmitted as a serialized message via base servers. Because the choice is determined automatically, programmers need not take care of memory environments.

See Appendix C for the details of the message protocol in the Tascell framework.

```
task pentomino {
  out: int s;
  in: int k, i0, i1, i2;
  in: int a[12]; // manage unused pieces
  in: int b[70]; // board
};
task_exec pentomino {
 this.r = search(this.k,this.i0,this.i1,this.i2,&this);
}

const int ps[57][5] = {...};  // see Figure 6.2
const int pos[13] = {...};    // see Figure 6.2

worker int search (int k, int j0, int j1, int j2, task pentomino *tsk) {
  int s=0; // the number of solutions
  // parallel for construct in Tascell
  for (int p : j1, j2)
    {
      int ap=tsk->a[p];
      for (int i=pos[ap]; i<pos[ap+1] ; i++) {
        // examine the "i-th" (piece, direction)
        ... local variable definitions ...
        if (Can ps[i] be set into the board tsk->b?);
        else continue;
        dynamic_wind // construct for specifying undo/redo operations
        { // do/redo operation for dynamic_wind
          Set the piece ps[i] into the board tsk->b and update tsk->a.
        }
        { // body for dynamic_wind
          kk = the next empty cell;
          if (no empty cell?) s++;      // a solution found
          else                          // the next piece
            s += search (kk, j0+1, j0+1, 12, tsk);
        }
        { // undo operation for dynamic_wind
          Remove the piece from tsk->b and restore tsk->a (backtracking).
        } // end of dynamic_wind
      }
    }
  pentomino (i1, i2) { // Declaration of this and setting a range (i1--i2) is done implicitly
    // put part (performed before sending a task)
    { // put task inputs for upper half iterations
      copy_piece_info (this.a, tsk->a);
      copy_board (this.b, tsk->b);
      this.k=k; this.i0=j0; this.i1=i1; this.i2=i2;
    }
    // get part (performed after receiving the result)
    { s += this.s; }
  }  // end of parallel for
  return s;
}
```

Figure 6.9: A Tascell Program for Pentomino.

6.4.2 Tascell Language

The Tascell language is an extended C language.[1] Figures 6.8 and 6.9 are examples of Tascell programs.

Programmers can write a worker program with new constructs in Tascell, starting with an existing sequential program. Tascell has constructs for defining a task and for specifying potential task division possibly with temporary undo/redo operations.

Task definition

A top-level task declaration:

`task` *task-name* `{` [`in:`|`out:`] *struct-declaration* ⋯ `};`

gives the structure of *task-name* task objects. For instance, "`task tfib {in: int n; out: int r};`" in Figure 6.8 declares the structure of `task tfib` objects. The syntax is the same as definitions of `struct`s, except that we may specify an `in:` or `out:` attribute for each field. A Tascell compiler uses attributes to construct default send/receive methods of the task. In addition, we can add user-defined send/receive methods in order to skip transmitting (part of) inputs/outputs selectively or allocating/freeing a working space (the details are omitted in this paper due to space limitation).

Definitions of entry points A top level declaration

`task_exec` *task-name* `{` *body* `}`

defines the computation of a *task-name* task. In the *body*, we can refer to the task object by the keyword `this`, which includes an input of the task in some fields, and we should set the result of the computation into appropriate fields. For instance, "`task_exec tfib {this.r=fib(this.n);}`" in Figure 6.8 assigns the result of `fib(this.n)` into `this.r`.

[1] Our actual Tascell language has an S-expression based syntax [18], but we write programs with a C-like syntax here for readers' convenience.

Definitons of send/receive methods for inter-node communication There are default send/receive methods automatically defined for each tasks, which send/receive values of fields with `in:`/`out:` attributes; fields with `in:` are sent/received by default task send/receive methods and fields with `out:` are sent/received by default result send/receive methods. In addition, we can add user-defined send/receive methods, which are called after default send/receive methods, in order to skip transmitting (part of) inputs/outputs selectively or allocating/freeing a working space.

A top-level declaration:

`task_receive` *task-name* `{` *body* `}`

defines how to receive serialized data from an input stream to initialize a task object. This receiver called when a worker receives a `task` message for *task-name* from another remote worker. We can refer to the task object (which includes parameters) by the keyword `this` and use library functions to receive values of basic types from the input stream (e.g., `recv_int`).

A top-level declaration:

`task_send` *task-name* `{` *body* `}`

defines how to send serialized data to an output stream to send a task object. This receiver is called when a worker sends a `task` message for *task-name* to another remote worker. We can refer to the task object by the keyword `this` and use library functions to send values of basic types to the output stream (e.g., `send_int`).

A result sender/receiver is defined in the same way as a task sender/receiver:

`rslt_receive` *task-name* `{` *body* `}`
`rslt_send` *task-name* `{` *body* `}`.

Each of these is called after/before a worker receives/sends a `rslt` message from/to another remote worker.

Definitons of send/receive methods for intra-node communication Transferring a task object between workers in the same computation node only requires passing a pointer to the task object by default. However, in some cases, we need some side effect such as allocating/freeing a working space. Such operations can be specified as local transfer methods:

`local_task_transfer` *task-name* `{` *body* `}`
`local_rslt_transfer` *task-name* `{` *body* `}`.

Constructs for task division

A statement:

`do_two` $statement_1$ $statement_2$
task-name `{` $statement_{put}$ $statement_{get}$ `}`

indicates that a computation in $statement_2$ ("`fib(n-2)`" in Figure 6.8) may be spawned during the execution of $statement_1$ ("`s1=fib(n-1);`" in Figure 6.8). More precise steps are as follows:

1 $Statement_1$ is executed with an implicit *task request handler*. If this handler is invoked with a task request, it divides the current task by spawning a new *task-name* task, setting the fields of the new task object by $statement_{put}$ ("`this.n=n-2;`" in Figure 6.8), and then sending it to the task requester. Here, a computation in $statement_2$ is packed as the task object.

2a If the task request handler for `do_two` is not invoked during the execution of $statement_1$, $statement_2$ is then executed.

2b Otherwise, the worker skips $statement_2$, waits for the result of the spawned task and then merges the result by executing $statement_{get}$ ("`s2=this.r;`" in Figure 6.8). In waiting for the result, the worker may execute another task taken back from the task requester in the Step 1 (by sending back a task request).

The identifier *task-name* specifies the type of a task to be created. The keyword `this` can be used in $statement_{put}$ and $statement_{get}$ to refer to the task object. We should initialize a task in $statement_{put}$ by assigning values to input fields, and can get a result of the task in $statement_{get}$ by referring to output fields. This series of operations should be equivalent to $statement_2$.

For dividing an iterative computation, Tascell has the parallel `for` loop construct syntactically denoted by:

`for(int` *identifier* `:` $expression_{from}$`,` $expression_{to}$`)`
  $statement_{body}$
task-name `(int` $identifier_{from}$`,` `int` $identifier_{to}$`)`
`{` $statement_{put}$ $statement_{get}$`}`

For example, Figure 6.9 employs a parallel `for` statement of "`for (int p: j1, j2) {...} pentomino (int i1,int i2) {{...} {s+=this.s}}`." This iterates $statement_{body}$ over integers from $expression_{from}$ (inclusive) to $expression_{to}$ (exclusive). When the implicit task request handler (available during the iterative execution of $statement_{body}$) is invoked, the upper half of remaining iterations are spawned as a new *task-name* task. The actual assigned range can be referred to in $statement_{put}$ by $identifier_{from}$ and $identifier_{to}$. The worker handles the result of the spawned task by executing $statement_{get}$.

Tascell has the `dynamic_wind` construct as in the Scheme language [25] to define undo/redo operations, syntactically denoted by:

`dynamic_wind` $statement_{before}$ $statement_{body}$ $statement_{after}$.

This basically executes $statement_{before}$("set a piece" in Figure 6.9 as "do"), $statement_{body}$ and $statement_{after}$("remove the piece" in Figure 6.9 as "undo") in this order. During the execution of $statement_{body}$, however, $statement_{after}$ is also executed as an "undo" clause *before* an attempt to invoke an older task request handler. $Statement_{before}$ is also executed as a "redo" clause *after* the attempt.

Backtracking-based task division `Do_two`, parallel `for` and `dynamic_wind` statements may be nested *dynamically* in their $statement_1$ or $statement_{body}$. Therefore, multiple task request handlers and undo-redo clauses may be available at the same time as in Figures 6.5 and 6.6. Each worker tries to detect a task request by polling at every `do_two` or parallel `for` statement. When the worker detects a task request, it performs temporary backtracking in order to spawn a larger task by invoking as old a handler as possible. If there are undo-redo clauses on the backtracking path, undo clauses are executed in turn for the backtracking and redo clauses are executed in turn for the restart.

Tascell Programming

We can write the Tascell program in Figure 6.8 by (1) starting with the C program in Figure 6.1, (2) adding the keyword `worker` to the procedure `fib`, (3) finding two statements which can be executed in parallel, (4) forming a `do_two` statement with the consideration of the name and structure of the spawned task, and (5) defining the structure and body of the task.

We can write the Tascell program in Figure 6.9 by starting with Figure 6.2 as above, except that (1) we find iterations which can be executed in parallel if separate working spaces are supplied, (2) we form a parallel `for` statement, (3) we prepare some working space in the task structure and adjust the access to it, (4) we form a `dynamic_wind` statement with existing do/undo operations, and (5) we adjust the parameter and body of `search` in order to accept a task with iterations. Notice that this program avoids undesirable copies of working spaces and promotes reuse/sharing of the working space.

6.5 Implementation

We implemented a Tascell compiler as a translator to standard C, which realizes high portability. To realize the backtracking-restarting mechanism, accessing "sleeping" variables (variables whose values are located below the current frame in the execution stack) is needed. Therefore, it seems to be impossible to implement this feature in standard C. But this problem is solved by the LW-SC language as described in Chapter 4. That is, we can get the translator to C by additionally implementing a translator from Tascell

```
int fib(void (*_bk0) lightweight (void), struct thread_data *_thr, int n)
{
  if (n <= 2)
    return 1;
  else {
    int s1, s2;
    { /*------------------ do_two ------------------*/
      struct tfib pthis[1];   // working space
      int spawned = 0;        // statement2 is spawned?
      {
        void _bk1_do_two lightweight (void)//nested function
        {
          if (spawned) return;
          _bk0(); // continue backtracking
          if (task request exists?) {
            pthis->n = n - 2;    //statement_put
            spawned = 1;
            make_and_send_task(_thr, 0, pthis); // spawn
          }
        }
        if (_thr->req)    // polling
          _bk1_do_two (); // start backtracking (call the nested function defined above)
        {
          s1 = fib(_bk1_do_two, _thr, n-1); // statement1
        }
      }
      if (spawned) {
        // Get and integrate the result of the spawned task
        wait_rslt(_thr);
        s2 = pthis->r;                // statement_get
      } else {
        s2 = fib(_bk0, _thr, n - 2);   // statement2
      }
    } /*------------------ do_two ------------------*/
    return s1 + s2;
  }
}
```

Figure 6.10: The translation result from the worker function `fib` in Figure 6.8, including translation of a `do_two` statement.

into LW-SC.

The program in Figure 6.8 is translated to the program in Figure 6.10 with nested functions. Each worker function is translated to have an additional parameter `_bk0` holding a nested function pointer corresponding to the newest handler for `do_two`, parallel `for` or `dynamic_wind` statements. Each `do_two` statement is translated into a piece of code which includes a definition of a nested function (`_bk1_do_two` in Figure 6.10) as the newest handler, which is called when a task request is detected by polling. The nested function first tries to spawn a larger task by calling a nested function (`_bk0`) which corresponds to the second newest handler (which calls another nested function for the third newest handler and so on). Only if a task request still remains, a new task is created and sent to the requester. After sending a task, the worker returns from the

```
int search (void (*_bk0) lightweight (void), struct thread_data *_thr,
            int k, int j0, int j1, int j2, struct pentomino *tsk)
{ int s = 0;                // the number of solutions
  { /*------------------ parallel for ------------------*/
    int p = j1;    int p_end = j2;
    struct pentomino *pthis;
    int spawned = 0;                                    // the number of spawned tasks
    void _bk1_par_for lightweight (void){ //nested function
      if (!spawned) _bk0();                             // continue backtracking
      while (p + 1 < p_end && task request exists?) {
        int i1 = (1 + p + p_end)/2, i2 = p_end;         // the range for the sub-task
        p_end = i1;                                     // shrink the range for itself
        pthis = malloc(sizeof(struct pentomino));       // allocate a working space
        { //statement_put
          copy_piece_info(pthis->a, tsk->a);
          copy_board(pthis->b, tsk->b);
          pthis->k = k; pthis->i0 = j0; pthis->i1 = i1; pthis->i2 = i2; }
        spawned++;
        make_and_send_task(_thr, 0, pthis); // spawn
      }
    }
    if (_thr->req)      // polling
      _bk1_par_for();  // start backtracking (call the nested function defined above)
    for (; p < p_end; p++) {
      int ap = (tsk->a)[p];
      for (int i = pos[ap]; i < pos[ap + 1]; i++) {
        // examine the ''i-th'' (piece, direction)
        ... local variable definitions ...
        if (Can ps[i] be set into the board tsk->b?);
          else continue;
        { /*------------ dynamic_wind ------------*/
          { // do operation (statement_before)
            Set the piece ps[i] into the board tsk->b and update tsk->a. }
          {
            void _bk2_dwind lightweight(void)
            { // nested function
              { // undo operation (statement_after)
                  Remove the piece from tsk->b and restore tsk->a (backtracking). }
              _bk1_par_for(); // continue backtracking (call the nested function defined above)
              { // redo operation (statement_before)
                Set the piece ps[i] into the board tsk->b and update tsk->a. }
            }
            { // statement_body
              kk = the next empty cell;
              if (no empty cell?) s++;                                  // a solution found
              else s += search (_bk2_dwind, _thr, kk, j0+1, j0+1, 12, tsk); // the next piece
            }
          }
          { // undo operation (statement_after)
            Remove the piece from tsk->b and restore tsk->a (backtracking). }
        } /*------------ dynamic_wind ------------*/
    } }
    while (spawned-- > 0) {
      // Get and integrate results of spawned tasks
      pthis = (struct pentomino *)wait_rslt(_thr);
      s += pthis->s;        // statement_get
      free(pthis); }
  } /*------------------ parallel for ------------------*/
  return s;
}
```

Figure 6.11: The translation result from the worker function `search` for Pentomino in Figure 6.9, including translation of a parallel `for` statement and a `dynamic_wind` statement.

nested function and resumes its own computation.

A parallel `for` statement can be translated in the same way (Figure 6.11), except that, in the nested function, the worker needs to calculate a ranges for a new task and update a range for itself.

Translation for a `dynamic_wind` statement is also included in Figure 6.11. As you can see, $statement_{body}$ employs a nested function as the newest one, which is composed of (a copy of) $statement_{after}$ (as undo operations), a call to the second newest nested function, and (a copy of) $statement_{before}$ (as redo operations), in order to perform undo/redo operations as is described in Section 6.4.2.

6.6 Related work

LTC is one of the best implementation techniques for dynamic load balancing. In LTC, a newly created thread is directly and immediately executed like a usual call while (the continuation of) the oldest thread in the computing resource may be stolen by other idle computing resources. Usually, the idle computing resource (*thief*) randomly selects another computing resource (*victim*) for stealing a task. A message passing implementation [7] of LTC employs a polling method where the victim detects a task request sent by the thief and returns a new task created by splitting the present running task. OPA [49] and StackThreads/MP [47] employ this technique.

Tascell is similar to WorkCrews [51] and Lazy RPC [8]. They take the parent-first strategy; at fork point, a worker executes the parent thread prior to the child thread and makes the child stealable for other workers, and calls the child thread if it has not been stolen at the join point of the parent thread.

Tascell supports distributed memory environments by transmitting inputs and outputs as task objects among computation nodes, instead of using distributed shared memory which distributed Cilk [36] and SilkRoad [31] use. Our approach enables programmers to program without consideration of the difference between shared and distributed memory environments because the interface for passing task objects is integrated. Furthermore, it is easy for new computation nodes to join a running computation dynamically.

6.7 Evaluation

In this section, we evaluate the performance of the Tascell framework using the following programs:

Fib(n) recursively computes the n-th Fibonacci number.

Nqueens(n) finds all solutions to the n-queens problem.

Pentomino(n) finds all solutions to the Pentomino problem with n pieces (using additional pieces and an expanded board for $n > 12$).

LU(n) computes the LU decomposition of an $n \times n$ matrix.

Comp(n) compares array elements a_i and b_j for all $0 \leq i, j < n$.

Grav(n) computes a total force exerted by $(2n + 1)^3$ uniform particles.

LU and Comp use a cache-oblivious recursive algorithm.

The evaluation environment is summarized as follows:

- the Tascell server
  - CPU: AMD Opteron 244 1.8GHz[2]
  - OS: Rocks 4.0 (Linux kernel 2.6.9)
  - Allegro Common Lisp 8.1 with `(speed 3) (safety 1) (space 1)` optimizers
- Computation nodes
  - CPU: AMD Dual Core Opteron 265 1.8GHz[2]×2 (4 cores in total)
  - OS: Rocks 4.0 (Linux kernel 2.6.9)
  - GCC 3.4.3 with `-O2` optimizers

[2]Optimized Power Management (OPM) is invalidated in order to prevent cores' frequencies from varying.

	Elapsed time in seconds (relative time to plain C)			
	C	Cilk	Tascell	Tascell (w/ copying)
Fib(40)	0.926 (1.00)	7.15 (7.72)	2.30 (2.48)	—
Nqueens(15)	10.3 (1.00)	29.8 (2.89)	15.8 (1.53)	25.4 (2.47)
Pentomino(12)	1.68 (1.00)	2.37 (1.41)	2.19 (1.30)	2.80 (1.67)
LU(2000)	8.66 (1.00)	8.54 (0.986)	8.69 (1.00)	—
Comp(30000)	4.31 (1.00)	7.84 (1.82)	5.42 (1.26)	—
Grav(200)	3.35 (1.00)	7.01 (2.09)	4.55 (1.35)	—

Table 6.1: Performance measurements with one worker.

- Network (in a distributed memory environment)
  - Gigabit Ethernet
  - Each node is TCP/IP connected to the single Tascell server.

To evaluate serial overheads, we ran the Tascell programs with one worker and compared their execution time with C and Cilk (version 5.3) programs in the same algorithms.

In Nqueens, Pentomino and Grav programs, each thread requires its own working space to hold one or more arrays. This is the case in many multithreaded languages other than Cilk. In Cilk, a pseudo variable `SYNCHED` is provided, which promotes the reuse of a working space among child logical threads [44], but child threads cannot share a working space with their parent thread. Furthermore, for Nqueens and Pentomino, each thread needs its own copy of its parent thread's working space, resulting in considerable copying overhead. In Tascell, the worker can reuse a single working space while it performs a sequential computation as is shown in Section 6.3.

The results of the performance measurements are shown in Table 6.1. The overheads in Tascell, which arise from polling and managing nested functions, are much lower than Cilk for almost all applications. In particular, Fib shows a sharp contrast in overheads because frequent creation of logical threads caused a higher overhead in Cilk. Nqueens shows a higher contrast than Pentomino because of more frequent copying. LU shows little overheads in both Cilk and Tascell because potential task division is infrequent.

There are additional overheads in Cilk that are broken down as follows:

(a) cost for explicit frame management,

(b) for the *THE* protocol [10] for consistent access to the logical thread queue, and

(c) cost of copying between working spaces for each thread (for Pentomino and Nqueens).

The copying overhead can be estimated as the difference between Tascell programs with and without artificial copying shown in Table 6.1.

Table 6.2 shows the results of performance measurements with multiple workers in a shared memory environment. Tascell almost always shows higher performance due to its lower serial overheads. For instance, we achieved a speedup of 1.86 times (= 49.9s/26.8s) as compared with Cilk in Nqueens(16). Though both Tascell and Cilk show good speedups, Tascell's speedups are lower. This is because the current implementation uses `pthread_cond_wait` for intra-node communication. (We expect that we can improve by using shared-memory operations more directly.)

Tascell shows super-linear speedups in Comp because of the larger accumulated cache size.

Tables 6.3 and 6.4 show the results of performance measurements on multiple computation nodes, and Figures 6.12 and 6.13 show log-scaled graphs, which correspond to Table 6.3, Table 6.4 respectively.[3]

In the environments where one worker running in each node (Table 6.3 and Figure 6.12), Fib, Nqueens, Pentomino, Comp and Grav show good speedups because their computation time is sufficiently long relative to the amount of transmission among computation

[3] We evaluated only Tascell because the standard implementation of Cilk only supports shared-memory environments.

Elapsed time in seconds
(speedup relative to one worker execution time)

	$T_{(1,1)}$	$T_{(2,1)}$	$T_{(4,1)}$
Fib(40)	2.30	1.29	0.677
	(1.00)	(1.78)	(3.40)
Fib(42)	6.04	3.40	1.73
	(1.00)	(1.78)	(3.49)
Fib(44)	15.8	8.43	4.46
	(1.00)	(1.87)	(3.54)
Nqueens(15)	15.8	7.95	4.02
	(1.00)	(1.99)	(3.93)
Nqueens(16)	107	53.5	26.8
	(1.00)	(2.00)	(3.99)
Pentomino(12)	2.19	1.15	0.579
	(1.00)	(1.90)	(3.78)
Pentomino(13)	17.6	8.82	4.51
	(1.00)	(2.00)	(3.90)
LU(2000)	8.69	4.55	2.55
	(1.00)	(1.91)	(3.41)
Comp(30000)	5.42	2.61	1.35
	(1.00)	(2.08)	(4.01)
Comp(60000)	21.8	10.1	5.14
	(1.00)	(2.16)	(4.24)
Grav(200)	4.55	2.31	1.41
	(1.00)	(1.97)	(3.23)

(a) Tascell

	$T_{(1,1)}$	$T_{(2,1)}$	$T_{(4,1)}$
Fib(40)	7.15	3.56	1.78
	(1.00)	(2.01)	(4.02)
Fib(42)	18.8	9.29	4.66
	(1.00)	(2.02)	(4.03)
Fib(44)	49.1	24.3	12.12
	(1.00)	(2.02)	(4.05)
Nqueens(15)	29.8	14.9	7.47
	(1.00)	(2.00)	(3.99)
Nqueens(16)	199	99.9	49.9
	(1.00)	(1.99)	(3.99)
Pentomino(12)	2.37	1.19	0.602
	(1.00)	(1.99)	(3.94)
Pentomino(13)	21.4	10.8	5.41
	(1.00)	(1.98)	(3.96)
LU(2000)	8.54	4.47	2.50
	(1.00)	(1.91)	(3.42)
Comp(30000)	7.84	4.07	2.04
	(1.00)	(1.93)	(3.84)
Comp(60000)	30.9	16.1	7.94
	(1.00)	(1.92)	(3.89)
Grav(200)	7.00	3.53	1.79
	(1.00)	(1.98)	(3.91)

(b) Cilk

Table 6.2: Execution time $T_{(k,1)}$ (and relative speedup) with k workers in a shared memory environment within one node.

	Elapsed time in seconds (speedup relative to one worker execution time)				
	$T_{(1,1)}$	$T_{(1,2)}$	$T_{(1,4)}$	$T_{(1,8)}$	$T_{(1,16)}$
Fib(40)	2.30	1.30	0.707	0.411	0.244
	(1.00)	(1.77)	(3.25)	(5.60)	(9.42)
Fib(42)	6.04	3.36	1.77	0.940	0.515
	(1.00)	(1.80)	(3.41)	(6.43)	(11.7)
Fib(44)	15.8	8.58	4.57	2.10	1.23
	(1.00)	(1.84)	(3.46)	(7.52)	(12.8)
Nqueens(15)	15.8	7.99	4.09	2.10	1.16
	(1.00)	(1.98)	(3.86)	(7.52)	(13.6)
Nqueens(16)	107	53.7	27.0	13.7	7.02
	(1.00)	(1.99)	(3.96)	(7.81)	(15.2)
Pentomino(12)	2.19	1.19	0.664	0.352	0.331
	(1.00)	(1.78)	(3.19)	(6.02)	(6.40)
Pentomino(13)	17.6	8.87	4.79	2.52	1.44
	(1.00)	(1.98)	(3.67)	(6.98)	(12.2)
LU(2000)	8.69	46.2	52.2	54.0	55.1
	(1.00)	(0.188)	(0.166)	(0.161)	(0.158)
Comp(30000)	5.42	2.57	1.34	0.955	0.786
	(1.00)	(2.11)	(4.04)	(5.68)	(6.90)
Comp(60000)	21.8	10.0	5.45	3.07	2.03
	(1.00)	(2.18)	(4.00)	(7.10)	(10.7)
Grav(200)	4.55	2.31	1.17	0.643	0.377
	(1.00)	(1.97)	(3.89)	(7.08)	(12.1)

Table 6.3: Elapsed time $T_{(1,\ell)}$ (and relative speedup) with ℓ distributed nodes each using one workers.

	Elapsed time in seconds (speedup relative to one worker execution time)				
	$T_{(4,1)}$	$T_{(4,2)}$	$T_{(4,4)}$	$T_{(4,8)}$	$T_{(4,16)}$
Fib(40)	0.677	0.369	0.229	0.178	0.137
	(3.40)	(6.23)	(10.0)	(12.9)	(16.8)
Fib(42)	1.73	0.912	0.552	0.354	0.257
	(3.49)	(6.62)	(10.9)	(17.1)	(23.5)
Fib(44)	4.46	2.55	1.24	0.789	0.534
	(3.54)	(6.20)	(12.7)	(20.0)	(29.6)
Fib(50)	76.1	39.2	19.9	10.6	6.80
	(3.73)	(7.24)	(14.3)	(26.8)	(41.8)
Nqueens(15)	4.02	2.06	1.12	0.736	0.532
	(3.93)	(7.67)	(14.1)	(21.5)	(29.7)
Nqueens(16)	26.8	13.5	7.09	3.96	2.41
	(3.99)	(7.93)	(15.1)	(27.0)	(44.4)
Nqueens(17)	190	95.3	49.4	25.0	13.4
	(3.98)	(7.94)	(15.3)	(30.3)	(56.5)
Pentomino(12)	0.579	0.379	0.235	0.233	0.213
	(3.78)	(5.78)	(9.32)	(9.34)	(10.3)
Pentomino(13)	4.51	2.64	1.46	0.931	0.772
	(3.90)	(6.67)	(12.1)	(18.9)	(22.3)
Pentomino(14)	38.7	22.8	11.4	6.28	4.48
	(3.98)	(6.75)	(13.5)	(24.5)	(34.4)
LU(2000)	2.55	14.2	29.2	40.3	36.1
	(3.41)	(0.612)	(0.298)	(0.216)	(0.241)
Comp(30000)	1.35	0.749	0.509	0.495	0.580
	(4.01)	(7.23)	(10.6)	(10.9)	(9.34)
Comp(60000)	5.14	2.72	1.85	1.43	1.30
	(4.24)	(8.01)	(11.8)	(15.2)	(16.8)
Grav(200)	1.41	0.756	0.445	0.361	0.253
	(3.23)	(6.02)	(10.2)	(12.6)	(18.0)
Grav(400)	10.2	5.69	2.97	1.85	1.08
	(3.55)	(6.36)	(12.2)	(19.6)	(33.5)

Table 6.4: Elapsed time $T_{(4,\ell)}$ (and relative speedup) with ℓ distributed nodes each using 4 workers.

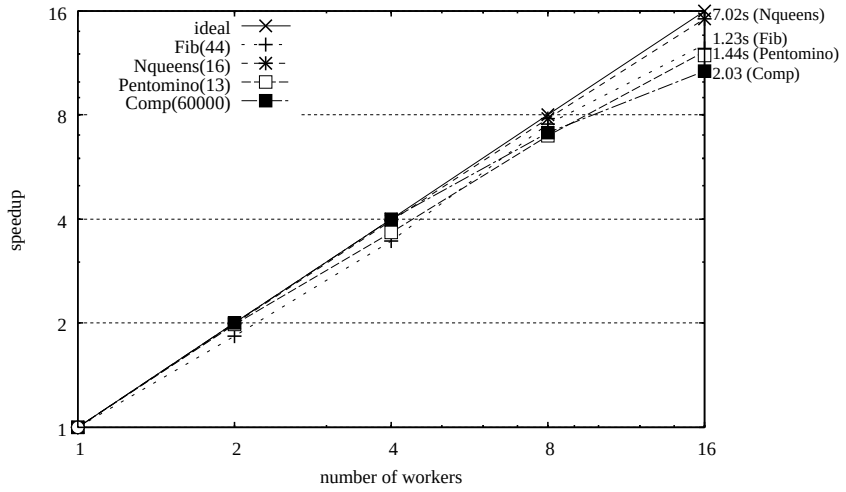


Figure 6.12: Speedups with multiple computation nodes each using one worker (corresponding to Table 6.3)

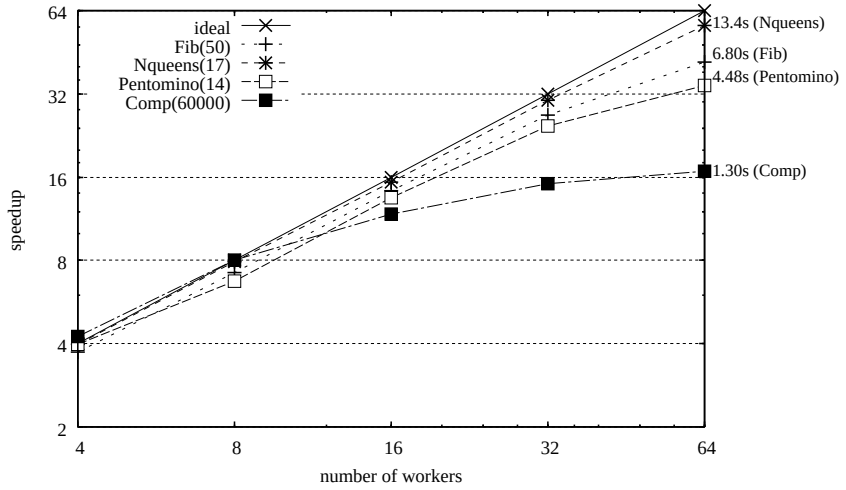


Figure 6.13: Speedups with multiple computation nodes each using 4 workers (corresponding to Table 6.4)

nodes. In contrast, we could not get speedups in LU because the communication overheads for transmitting sub-matrices as sub-tasks or sub-results are much larger than the effect of parallelization. Generally, it is difficult to obtain sufficient speedups in applications with huge shared data such as LU with fully dynamic load balancing, as experienced also in [50].

In environments where multiple workers running in each node, the workers run with both inter- and intra-node communication. Table 6.4 and Figure 6.13 show that we can get a good speedup also in such an environment as long as the size of a problem is sufficiently large relative to the number of workers. The speedups in Comp are limited because transmission costs of $O(n)$ do not pay for small n since the time complexity of Comp is $O(n^2)$.

We can improve performance in distributed memory environments by improving message handling of a Tascell server or employing some mechanism for sharing data among computation nodes.

Chapter 7

Related Work

This chapter introduces related work that is relevant with the SC language system and our strategy for language extensions. See also Section 4.7, Section 5.4 and Section 6.6 for related work about each topic of the corresponding chapter.

7.1 Language Extensions by Code Translation

There exist many useful extensions to C such as Cilk [10] and OpenMP [32], but their purpose is to implement their own extensions, not to make a framework for general language extensions.

Lisp/Scheme is easy to be extended by transformation and to be written by humans. Furthermore, there exist compilers from their languages to C [60, 22]. But they are not suitable for describing low level operations such as pointer operations. From another point of view, the SC language system applies the advantages of Lisp in extensibility to C with preservation of its ability of low-level operations.

7.2 Lower-Level Scheme

Pre-Scheme [24] is a dialect of Scheme, which lacks some features such as garbage collection and full proper tail-recursion but preserves the other Scheme's advantages such as higher order procedures, interactive debugging, and macros and gives low-level machine access of C. Our approach is to support language developers to implement language

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

7.3 Reflection

Reflection is to manipulate behaviors of a running program by referring to or modifying meta-level information as a first-class object, which enables us to extend a program dynamically. Although it is very powerful in extensibility, it causes great decrease of performance in many implementations that such information is kept by an interpreter. Most implementation [5] overcomes this problem by restricting the extension targets. *Compile-time reflection* [4, 46, 39, 38, 40] realizes such extension by transforming programs in compile-time, which is similar to our approach. But we provide more generic framework to transform programs such as from LW-SC into C.

7.4 Aspect Oriented Programming

Aspect Oriented Programming [26] (AOP) is a programming paradigm to handle a cross-cutting concern in one place. In general, it is implemented by inserting a method defined as an *advice* into *join points* specified in a program. The SC language system also can be used to implement this feature defining such insertion by adequate transformation rules.

7.5 Pattern-matching

There exist many implementations of pattern-matching utilities on S-expressions [34]. But the patterns which correspond to `,@`*symbol* and `,@`*symbol*`[`*function-name*`]` are not popular.

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

7.6 Another S-Expression Based C

Symbolic C Expressions [28] (Scexp) is another S-expression based C language. It emphasizes usefulness of writing C code with Lisp macros. However, it is not intended to be a base for language extensions.

7.7 Other Rule-based Transformations

7.7.1 Expert systems

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

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

7.7.2 Rewriting rules

There are program transformation systems that use rewrite rules [52, 53, 6]. In most such systems, rules are defined more declaratively and environments for object languages are included in both patterns and outputs, while our translators treat such environments basically by using side-effects or dynamic bindings in Common Lisp.

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

7.7.3 XML

XML [54] is often used as internal representation for code analysis and transformation [15]. Transformation over XML can be defined using XSLT (XSL Transformation) [55].

XML is also applicable for analysis and transformation, but does not suit our purpose because it is too complicated to be written by humans unlike S-expressions.

7.8 Programs as Lisp Macro Forms

Our system treats SC programs as data. On the other hand, there are some S-expression based markup languages where programs themselves are evaluated by a Lisp evaluator as macro forms and object code is generated as a result [37, 9].

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

Chapter 8

Conclusion and Future Work

We proposed a scheme for extending the C language, where an S-expression based extended language is translated into a C language with an S-expression syntax. In this scheme, we can extend C at lower implementation cost because we can easily manipulate S-expressions using Lisp and transformation rules can be written intuitively using pattern-matching over S-expressions.

We also presented a technique to implement nested functions for the C language, employing the SC language system. Since the implementation is transformation-based, it enables us to implement high-level services with "stack walk" in a portable way. Furthermore, such services can be efficiently implemented because we aggressively reduce the cost of creating and maintaining nested functions using "lightweight" closures.

In order to enhance usefulness of SC languages, we implemented a C-to-SC translator which enables SC programmers to include existing C header files. Since some C macros cannot be translated into SC, we discussed the limitations of the automatic translation and proposed practical countermeasure against them. Because a C macro can be defined using syntactically incomplete token strings as an expanded code, some macro cannot be translated into an SC macro which is defined using a parsing unit. But we can implement a practical translator using some hypotheses. This strategy can be applied when implementing interfaces between other languages. The translator also enables us to implement a translator for C programs by writing rule-sets for the SC language system.

We actually presented some language extensions using the SC language system and

nested functions featured by LW-SC. In particular, we proposed a practical example where a new scheme for dynamic load balancing, based on backtracking, is realized. This scheme realizes quite low serial overheads because of no use of logical threads as potential tasks and lazy allocation of working spaces. In particular, our scheme is useful for search problems which include various important applications such as combinatorial optimization problems. In addition, our task-object-based parallel programming model enables programs to be easily applied for both shared and distributed (and also hybrid) memory environments.

The SC language system can also be used for implementing languages other than extended SC languages (e.g., Scheme and Java), which we plan to do as future work.

We will also improve Tascell described in Chapter 6, in aspects of ease of use and efficiency. In particular, we will achieve more efficient computation in distributed memory environments by implementing more sophisticated message handling among computation nodes or utilizing distributed data sharing mechanisms such as DSM (Distributed Shared Memory). We will also implement a mechanism to enable computation nodes to leave safely.

Appendix A

The Syntax of the SC-1 Language

A.1 External Declarations

translation-unit :
    external-declaration
    translation-unit external-declaration

external-declaration :
    declaration

A.2 Declarations

declaration-list :
    declaration
    declaration-list declaration

declaration :
    inlined-declaration
    (*identifier type-expression initializer*$_{opt}$)
    (*function-identifier* (`fn` *function-type-list*)
      [`:attr` *function-attribute*]$_{opt}$*register-declarator*$_{opt}$ *block-item-list*$_{opt}$)
    (*struct-or-union-specifier struct-declaration-list*$_{opt}$)
    (*enum-specifier enumerator-list*)

inlined-declaration-list :
    inlined-declaration

inlined-declaration-list declaration

inlined-declaration :
(*storage-class-specifier identifier type-expression initializer*$_{opt}$)
(*storage-class-specifier function-identifier* (`fn` *function-type-list*)
[`:attr` *function-attribute*]$_{opt}$ *register-declarator*$_{opt}$ *block-item-list*$_{opt}$)
(*def-or-decl struct-or-union-specifier struct-declaration-list*$_{opt}$)
(`def` *enum-specifier enumerator-list*)
(*compound-storage-class-specifier type-expression init-declarator-list*)
(`deftype` *identifier type-expression*)
(`deftype` *identifier struct-or-union struct-declaration-list*$_{opt}$)
(`deftype` *identifier* `enum` *enumerator-list*)

function-identifier :
identifier
(*identifier-list*)

def-or-decl :
`def`
`decl`

init-declarator-list :
init-declarator
init-declarator-list init-declarator

init-declarator :
identifier
(*identifier initializer*)

storage-class-specifier : one of
`def decl extern extern-def extern-decl`
`static static-def auto auto-def register register-def`

compound-storage-class-specifier : one of
`defs extern-defs static-defs auto-defs register-defs`

function-attribute :
`inline`

register-declarator :

`(register` *identifier-list*`)`

struct-declaration-list :
struct-declaration
struct-declaration-list struct-declaration

struct-declaration :
declaration $[$`:bit` *expression*$]_{opt}$

enumerator-list :
enumerator
enumerator-list enumerator

enumerator :
enumeration-constant
`(`*enumeration-constant expression*`)`

enumeration-constant :
identifier

identifier-list :
identifier
identifier-list identifier

designator :
`(aref-this` *expression-list*`)`
`(fref-this` *identifier-list*`)`
`(aref` *designator expression-list*`)`
`(fref` *designator identifier-list*`)`

designated-initializer :
initializer
`(`*designator initializer*`)`

initializer-list :
designated-initializer
initializer-list designated-initializer

compound-initializer :
`(array` *initializer-list*`)`

`(struct` *initializer-list*`)`

initializer :
expression
compound-initializer

A.3 Type-expressions

type-expression :
type-specifier
`(`*type-qualifier-list type-expression*`)`
`(array` *type-expression array-subscription-list*$_{opt}$`)`
`(ptr` *type-expression*`)`
`(fn` *function-type-list*`)`

function-type-list :
type-expression-list `va-arg`$_{opt}$

type-expression-list
type-expression
type-expression-list type-expression

type-specifier : one of
`void`
`char signed-char unsigned-char short signed-short unsigned-short`
`int signed-int unsigned-int long signed-long unsigned-long`
`long-long signed-long-long unsigned-long-long`
`float double long-double`
struct-or-union-specifier
enum-specifier
typedef-name

array-subscription-list :
expression-list

struct-or-union-specifier :
`(`*struct-or-union identifier*`)`

struct-or-union :

`struct`
`union`

enum-specifier :
`(enum` *identifier*`)`

type-qualifier-list :
type-qualifier
type-qualifier-list type-qualifier

type-qualifier :
`const`
`restrict`
`volatile`

typedef-name :
identifier

A.4 Statements

statement :
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement
labeled-statement
`()`

compound-statement :
`(begin` *block-item-list*$_{opt}$`)`
`(let (`*declaration-list*$_{opt}$`)` *block-item-list*$_{opt}$`)`

block-item-list :
block-item
block-item-list block-item

block-item :
inlined-declaration

statement

labeled-statement :
`(label` *identifier statement*`)`
`(case` *expression*`)`
`(default)`

expression-statement :
expression

selection-statement :
`(if` *expression statement* $statement_{opt}$`)`
`(switch` *expression* $block\text{-}item\text{-}list_{opt}$`)`

iteration-statement :
`(while` *expression* $block\text{-}item\text{-}list_{opt}$`)`
`(do-while` *expression* $block\text{-}item\text{-}list_{opt}$`)`
`(for (`$expression\text{-}list_{opt}$ *expression expression*`)` $block\text{-}item\text{-}list_{opt}$`)`
`(for (`$inlined\text{-}declaration\text{-}list_{opt}$ *expression expression*`)` $block\text{-}item\text{-}list_{opt}$`)`
`(loop` $block\text{-}item\text{-}list_{opt}$`)`

jump-statement :
`(goto` *identifier*`)`
`(continue)`
`(break)`
`(return` $expression_{opt}$`)`

A.5 Expressions

expression :
identifier
constant
string-literal
compound-literal
`(`*expression-list*`)`
`(aref` *expression-list*`)`
`(fref` *expression field-identifier-list*`)`
`(inc` *expression*`)`
`(dec` *expression*`)`

(`++` *expression*)
(`--` *expression*)
(*unary-operator expression*)
(`sizeof` *expression*)
(`sizeof` *type-expression*)
(`cast` *type-expression expression*)
(*operator expression-list*)
(*comparator expression expression*)
(`if-exp` *expression expression expression*)
(*assignment-operator expression expression*)
(`exps` *expression-list*)

compound-literal :
(`init` *type-expression compound-initializer*)

expression-list :
expression
expression-list expression

field-identifier-list :
field-identifier
field-identifier-list field-identifier

field-identifier :
identifier
`->` *identifier*

operator : one of
`* / % + - << >> bit-xor bit-and bit-or and or`

comparator : one of
`< > <= >= == !=`

assignment-operator : one of
`= *= /= %= += -= <<= >>= bit-and= bit-xor= bit-or=`

unary-operator : one of
`ptr mref bit-not not`

Appendix B

An Example of Translation from LW-SC to SC-1

```
;;; The pointer to the moved ‘‘nested function’’.
(deftype nestfn-t
         (ptr (fn (ptr char) (ptr char) (ptr void))))
;;; The structure which contains the pointer to the moved
;;; nested function and the frame pointer of
;;; the owner function.
(deftype closure-t struct
  (def fun nestfn-t)
  (def fr (ptr void)))

(deftype align-t double)

;;; The auxiliary function for calling nested functions.
(def (lw-call esp) (fn (ptr char) (ptr char))
  (def clos (ptr closure-t)
            (mref (cast (ptr (ptr closure-t)) esp)))
  (return ((fref clos -> fun) esp (fref clos -> fr))))

;;; The frame structure of function h.
(def (struct h_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def tmp2 int)
  (def tmp int)
  (def g (ptr closure-t))
  (def i int))

(def (h esp i g)
     (fn int (ptr char) int (ptr closure-t))
  (def argp (ptr char))
  (def efp (ptr (struct h_frame)))
  (def new-esp (ptr char))
```

```
(def esp-flag size-t (bit-and (cast size-t esp) 3))
(def tmp int)
(def tmp2 int)
(def tmp_fp (ptr closure-t))
(def tmp_fp2 (ptr closure-t))
;; Judge whether reconstruction of the execution stack is
;; required or not.
(if esp-flag
    (begin
      (= esp (cast (ptr char)
                (bit-xor (cast size-t esp) esp-flag)))
      (= efp (cast (ptr (struct h_frame)) esp))
      ;; Move the stack pointer by the frame size.
      (= esp
         (cast (ptr char)
               (+ (cast (ptr align-t) esp)
                  (/ (+ (sizeof (struct h_frame))
                        (sizeof align-t) -1)
                     (sizeof align-t)))))
      (= (mref (cast (ptr (ptr char)) esp)) 0)
      ;; Restore the execution point.
      (label LGOTO
        (switch (fref (mref efp) call-id)
          (case 0) (goto l_CALL)
          (case 1) (goto l_CALL2)))
      (goto l_CALL)))
(= efp (cast (ptr (struct h_frame)) esp))
;; Move the stack pointer by the frame size.
(= esp
   (cast (ptr char)
         (+ (cast (ptr align-t) esp)
            (/ (+ (sizeof (struct h_frame))
                  (sizeof align-t) -1)
               (sizeof align-t)))))
(= (mref (cast (ptr (ptr char)) esp)) 0)
;; Call the nested function g.
(begin
  (= tmp_fp g)
  (= argp
     (cast (ptr char)
          (+ (cast (ptr align-t) esp)
             (/ (+ (sizeof (ptr char))
                   (sizeof align-t) -1)
                (sizeof align-t)))))
  ;; Push the arguments passed to nested function.
  (exps (= (mref (cast (ptr int) argp)) i)
        (= argp
           (cast (ptr char)
             (+ (cast (ptr align-t) argp)
                (/ (+ (sizeof int)
```

```
                    (sizeof align-t) -1)
                  (sizeof align-t))))))
  ;; Push the structure object that corresponds to
  ;; the frame of the nested function to
  ;; the explicit stack.
  (= (mref (cast (ptr (ptr closure-t)) argp)) tmp_fp)
  ;; Save the values of local variables to the frame.
  (= (fref efp -> tmp2) tmp2)
  (= (fref efp -> tmp) tmp)
  (= (fref efp -> g) g)
  (= (fref efp -> i) i)
  (= (fref efp -> argp) argp)
  (= (fref efp -> tmp-esp) argp)
  ;; Save the current execution point.
  (= (fref efp -> call-id) 0)
  (return (- (cast int 0) 1))
  ;; Continue the execution from here after the function call finishes.
  (label l_CALL nil)
  ;; Restore local variables from the explicit stack.
  (= tmp2 (fref efp -> tmp2))
  (= tmp (fref efp -> tmp))
  (= g (fref efp -> g))
  (= i (fref efp -> i))
  ;; Get the return value.
  (= tmp (mref (cast (ptr int) (fref efp -> argp)))))
;; Call the nested function g.
(begin
  (= tmp_fp2 g)
  (= argp
     (cast (ptr char)
       (+ (cast (ptr align-t) esp)
          (/ (+ (sizeof (ptr char))
                (sizeof align-t) -1)
             (sizeof align-t)))))
  ;; Push the arguments passed to nested function.
  (exps (= (mref (cast (ptr int) argp)) tmp)
        (= argp
           (cast (ptr char)
             (+ (cast (ptr align-t) argp)
                (/ (+ (sizeof int)
                      (sizeof align-t) -1)
                   (sizeof align-t))))))
  ;; Push the structure object that corresponds to
  ;; the frame of the nested function to
  ;; the explicit stack.
  (= (mref (cast (ptr (ptr closure-t)) argp))
     tmp_fp2)
  ;; Save the values of local variables to the frame.
  (= (fref efp -> tmp2) tmp2)
  (= (fref efp -> tmp) tmp)
  (= (fref efp -> g) g)
```

```
    (= (fref efp -> i) i)
    (= (fref efp -> argp) argp)
    (= (fref efp -> tmp-esp) argp)
    ;; Save the current execution point.
    (= (fref efp -> call-id) 1)
    (return (- (cast int 0) 1))
    ;; Continue the execution from here after
    ;; the function call finishes.
    (label l_CALL2 nil)
    (= tmp2 (fref efp -> tmp2))
    (= tmp (fref efp -> tmp))
    (= g (fref efp -> g))
    (= i (fref efp -> i))
    ;; Get the return value.
    (= tmp2 (mref (cast (ptr int)
                        (fref efp -> argp)))))
  (return tmp2))

;;; The frame structure of function foo.
(def (struct foo_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def tmp3 int)
  (def y int)
  (def x int)
  (def a int)
  (def g10 closure-t))

;;; The frame structure of function g1 .
(def (struct g1_in_foo_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def b int)
  (def xfp (ptr (struct foo_frame))))

;;; Nested function g1 (moved to the top-level).
(def (g1_in_foo esp xfp0)
     (fn (ptr char) (ptr char) (ptr void))
 (def new-esp (ptr char))
 (def efp (ptr (struct g1_in_foo_frame)))
 ;; The frame pointer of the owner function.
 (def xfp (ptr (struct foo_frame)) xfp0)
 (def esp-flag size-t (bit-and (cast size-t esp) 3))
 (def parmp (ptr char)
   (cast (ptr char)
         (bit-xor (cast size-t esp) esp-flag)))
 ;; Pop parameters from the explicit stack.
 (def b int
   (exps
    (= parmp
       (cast (ptr char)
```

```
            (- (cast (ptr align-t) parmp)
               (/ (+ (sizeof int) (sizeof align-t) -1)
                  (sizeof align-t)))))
   (mref (cast (ptr int) parmp))))
(label LGOTO nil)
(= efp (cast (ptr (struct g1_in_foo_frame)) esp))
;; Move the stack pointer by the frame size.
(= esp
   (cast (ptr char)
         (+ (cast (ptr align-t) esp)
            (/ (+ (sizeof (struct g1_in_foo_frame))
                  (sizeof align-t) -1)
               (sizeof align-t)))))
(= (mref (cast (ptr (ptr char)) esp)) 0)
(inc (fref xfp -> x))
;; Push the return value to the explicit stack.
(= (mref (cast (ptr int) efp)) (+ (fref xfp -> a) b))
(return 0))

(def (foo esp a) (fn int (ptr char) int)
  (def efp (ptr (struct foo_frame)))
  (def new-esp (ptr char))
  (def esp-flag size-t (bit-and (cast size-t esp) 3))
  (def x int 0)
  (def y int 0)
  (def tmp3 int)

  ;; Judge whether reconstruction of the execution stack is
  ;; required or not.
  (if esp-flag
      (begin
        (= esp (cast (ptr char)
                     (bit-xor (cast size-t esp) esp-flag)))
        (= efp (cast (ptr (struct foo_frame)) esp))
        ;; Move the stack pointer by the frame size.
        (= esp
           (cast (ptr char)
                 (+ (cast (ptr align-t) esp)
                    (/ (+ (sizeof (struct foo_frame))
                          (sizeof align-t) -1)
                       (sizeof align-t)))))
        (= (mref (cast (ptr (ptr char)) esp)) 0)
        (label LGOTO
          ;; Restore the execution point.
          (switch (fref (mref efp) call-id)
            (case 0) (goto l_CALL3)))
        (goto l_CALL3)))
  (= efp (cast (ptr (struct foo_frame)) esp))
  ;; Move the stack pointer by the frame size.
  (= esp
     (cast (ptr char)
```

```
          (+ (cast (ptr align-t) esp)
             (/ (+ (sizeof (struct foo_frame))
                   (sizeof align-t) -1)
                (sizeof align-t)))))
  (= (mref (cast (ptr (ptr char)) esp)) 0)
  (= new-esp esp)
  ;; Call the ordinary function h.
  (while
      (and
       (== (= tmp3 (h new-esp 10
                      (ptr (fref
                       (cast (ptr (struct foo_frame))
                             esp)
                       -> g10))))
           (- (cast int 0) 1))
       (!= (= (fref efp -> tmp-esp)
              (mref (cast (ptr (ptr char)) esp))) 0))
    ;; Save the values of local variables to the frame.
    (= (fref efp -> tmp3) tmp3)
    (= (fref efp -> y) y)
    (= (fref efp -> x) x)
    (= (fref efp -> a) a)
    (= (fref efp -> g10 fun) g1_in_foo)
    (= (fref efp -> g10 fr) (cast (ptr void) efp))
    ;; Save the current execution point.
    (= (fref efp -> call-id) 0)
    (return (- (cast int 0) 1))
    ;; Continue the execution from here after
    ;; the function call finishes.
    (label l_CALL3 nil)
    ;; Restore local variables from the explicit stack.
    (= tmp3 (fref efp -> tmp3))
    (= y (fref efp -> y))
    (= x (fref efp -> x))
    (= a (fref efp -> a))
    (= new-esp (+ esp 1)))
  (= y tmp3)
  (return (+ x y)))

;;; The frame structure of function main .
(def (struct main_frame)
  (def tmp-esp (ptr char))
  (def argp (ptr char))
  (def call-id int)
  (def tmp4 int))

(def (main) (fn int)
  (def efp (ptr (struct main_frame)))
  (def new-esp (ptr char))
  (def estack (array char 65536))  ; The explicit stack.
  (def esp (ptr char) estack)
  (def tmp4 int)
```

```
(label LGOTO nil)
(= efp (cast (ptr (struct main_frame)) esp))
;; Move the stack pointer by the frame size.
(= esp
   (cast (ptr char)
         (+ (cast (ptr align-t) esp)
            (/ (+ (sizeof (struct main_frame))
                  (sizeof align-t) -1)
               (sizeof align-t)))))
(= (mref (cast (ptr (ptr char)) esp)) 0)
(= new-esp esp)
(while
    (and
     (== (= tmp4 (foo new-esp 1))
         (- (cast int 0) 1))
     (!= (= (fref efp -> tmp-esp)
            (mref (cast (ptr (ptr char)) esp))) 0))
  (def goto-fr (ptr char))
  (= (mref (cast (ptr (ptr char)) esp)) 0)
  (= (fref efp -> tmp4) tmp4)
  ;; Execute nested functions.
  (= goto-fr (lw-call (fref efp -> tmp-esp)))
  (if (== (cast (ptr char) goto-fr)
          (cast (ptr char) efp))
      (goto LGOTO))
  (= new-esp (+ esp 1)))
(return tmp4))
```

Appendix C

Message Protocol in Tascell Framework

A message transmitted among workers and Tascell servers is one of the followings:

- `treq` $address_{src}$ $address_{dest}$ |`any`
  is generated by an idle worker to request a task to another worker. The worker which received this message sends back a `task` if it can spawn a new task, or a `none` otherwise. When a Tascell server received a `treq` with `any` as second argument, it asks a task of each computation node, and then the computation node asks a task of each worker. If no computation nodes have any task to be spawned, the server preserves the `treq` message and sends again it later.

- `task` *ndiv* $address_{src}$ $address_{dest}$ *task-no* [*data*]
  is generated by a worker which received a `treq` and has a task to be spawned. The *data* is serialized data of the task object, which is sent only if the message is for another computation node; the object is transmitted via a shared memory instead for the same computation node. The parameter *task-no* specifies which kind of a task to be driven. The parameter *ndiv* specifies how many times the task is divided. The information gives an indication of how large a task the computation node has. Therefore it is stored in the Tascell server and used to decide to which node `treq` messages with `any` arguments are sent.

- `none` $address_{dest}$
  is generated by a worker which received a `treq` but has no task to be spawned.

- `rslt` $address_{dest}$ [*data*]
  is generated by a worker which finished the computation of the `task` message to return the result. The *data* is serialized data of the task object, which is sent only if the message is for another computation node (as well as a `task` message).

- `rack` $address_{dest}$
  is generated by a worker which received a `rslt` message to acknowledge that the worker surely received the result.

Each $address_{src}$ or $address_{dest}$ is specified as a path from the direct receivers of the message to the destination, which is specified by a sequence of integers or '`p`'s separated by '`:`'s. A Tascell server can identify a computation node or a child server connected to it by an integer, and a computation node can identify a worker in it by an integer, too. In addition, the character `p` is used to point a parent. The addresses are modified when the messages are relayed by servers.

For example, sending a `treq` from the worker (a) to the worker (b) in Figure 6.7 is done as follows

1. The worker (a) sends "`treq` n`:0 0:1`" the connecting server.

2. The server sends "`treq p:`n`:0 1`" to the computation node of (b).

3. The computation node notify the worker (b) of the task request.

Then the worker (b) can return a `task` or a `none` to (a) by using the relative address "`p:`n`:0`".

Bibliography

[1] Bobrow, D. G., Mittal, S. and Stefik, M. J.: Expert systems: perils and promise, *Commun. ACM*, Vol. 29, No. 9, pp. 880–894 (1986).

[2] Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice & Experience*, Vol. 18, No. 9, pp. 807–820 (1988).

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

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

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

[6] Cleenewerck, T. and D'Hondt, T.: Disentangling the implementation of local-to-global transformations in a rewrite rule transformation system, *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, ACM Press, pp. 1398–1403 (2005).

[7] Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proceedings of the International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, No. 748, Springer-Verlag, pp. 94–107 (1993).

[8] Feeley, M.: Lazy Remote Procedure Call and its Implementation in a Parallel Variant of C, *Proceedings of International Workshop on Parallel Symbolic Languages and Systems*, Lecture Notes in Computer Science, No. 1068, Springer-Verlag, pp. 3–21 (1995).

[9] Franz Inc.: HTML Generation Facility. `http://allegroserve.sourceforge.net/aserve-dist/doc/htmlgen.html`.

[10] Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices (PLDI '98)*, Vol. 33, No. 5, pp. 212–223 (1998).

[11] Goldstein, S. C., Schauser, K. E. and Culler, D. E.: Lazy Threads: Implementing a Fast Parallel Call, *Journal of Parallel and Distributed Computing*, Vol. 3, No. 1, pp. 5–20 (1996).

[12] Graham, P.: *On LISP: Advanced Techniques for Common LISP*, Prentice Hall (1993).

[13] Halstead, Jr., R. H.: New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, *Parallel Lisp: Languages and Systems* (Ito, T. and Halstead, R. H., eds.), Lecture Notes in Computer Science, Vol. 441, Sendai, Japan, June 5–8, Springer, Berlin, pp. 2–57 (1990).

[14] Hanson, D. R. and Raghavachari, M.: A Machine-Independent Debugger, *Software – Practice & Experience*, Vol. 26, No. 11, pp. 1277–1299 (1996).

[15] Hayato, K. and Katsuhiko, G.: Designing Program Information Extraction System Based on ACML, *In Proc. of 20th National Convention JSSST* (2003).

[16] Henderson, F.: Accurate Garbage Collection in an Uncooperative Environment, *Proc. of the 3rd International Symposium on Memory Management*, pp. 150–156 (2002).

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

[18] Hiraishi, T., Yasugi, M. and Yuasa, T.: Experience with SC: Transformation-based Implementation of Various Language Extensions to C, *Proceedings of the International Lisp Conference*, Clare College, Cambridge, U.K., pp. 103–113 (2007).

[19] IEEE: *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6* (2001). Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[20] ISO/IEC: ISO/IEC 9899:1990(E) Programming Languages — C (1990).

[21] ISO/IEC: ISO/IEC 9899:1999(E) Programming Languages — C (1999).

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

[23] Jones, S. P., Ramsey, N. and Reig, F.: C−−: A Portable Assembly Language That Supports Garbage Collection, *International Conference on Principles and Practice of Declarative Programming* (1999).

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

[25] Kelsey, R., Clinger, W. and Rees, J.: Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices*, Vol. 33, No. 9, pp. 26–76 (1998).

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

[27] MacLachlan, R. A.: CMUCL User's Manual (2004). `http://www.cons.org/cmucl/`.

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

[29] Matsui, K.: MCPP — A portable C preprocessor with Validation Suite (2004). `http://www.m17n.org/mcpp/`.

[30] Mohr, E., Kranz, D. A. and Halstead, Jr., R. H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).

[31] Peng, L., Wong, W. F., Feng, M. D. and Yuen, C. K.: SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters, *IEEE International Conferrence on Cluster Computing (Cluster2000)*, pp. 243–249 (2000).

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

[33] Plevyak, J., Karamcheti, V., Zhang, X. and Chien, A. A.: A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers, *Supercomputing'95* (1995).

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

[35] Ramsey, N. and Jones, S. P.: A Single Intermediate Language That Supports Multiple Implementations of Exceptions, *Proc. of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pp. 285–298 (2000).

[36] Randall, K.: Cilk: Efficient Multithreaded Computing, Technical Report MIT/LCS/TR-749 (1998).

[37] Rosenberg, K.: LML: The Lisp Markup Language. `http://lml.b9.com/`.

[38] Roudier, Y. and Ichisugi, Y.: Integrating data-parallel and reactive constructs into Java, *In Proc. of France-Japan Workshop on OBPDC'97* (1997).

[39] Roudier, Y. and Ichisugi, Y.: Java Data-parallel Programming using an Extensible Java Preprocessor, *Swopp'97* (1997).

[40] Roudier, Y. and Ichisugi, Y.: Mixin Composition Strategies for the Modular Implementation of Aspect Weaving, *Aspect-Oriented Programming Workshop at ICSE'98* (1998).

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

[42] Steele Jr., G. L.: *Common Lisp: The Language,* Second Edition, Digital Press (1990).

[43] Strumpen, V.: Compiler Technology for Portable Checkpoints, `http://theory.lcs.mit.edu/~porch/` (1998).

[44] Supercomputing Technologies Group: *Cilk 5.4.6 Reference Manual*, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA.

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

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

[47] Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pp. 60–71 (1999).

[48] Taura, K. and Yonezawa, A.: Fine-Grain Multithreading with Minimal Compiler Support: A Cost Effective Approach to Implementing Efficient Multithreading Languages, *Proc. of Conference on Programming Language Design and Implementation*, pp. 320–333 (1997).

[49] Umatani, S., Yasugi, M., Komiya, T. and Yuasa, T.: Pursuing Laziness for Efficient Implementation of Modern Multithreaded Languages, *Proc. of the 5th International Symposium on High Performance Computing*, Lecture Notes in Computer Science, No. 2858, pp. 174–188 (2003).

[50] van Nieuwpoort, R. V., Kielmann, T. and Bal, H. E.: Efficient load balancing for wide-area divide-and-conquer applications, *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, New York, NY, USA, ACM, pp. 34–43 (2001).

[51] Vandevoorde, M. T. and Roberts, E. S.: WorkCrews: An Abstraction for Controlling Parallelism, *International Journal of Parallel Programming*, Vol. 17, No. 4, pp. 347–366 (1988).

[52] Visser, E.: Stratego: A Language for Program Transformation Based on Rewriting Strategies, *Lecture Notes in Computer Science*, Vol. 2051, pp. 357+ (2001).

[53] Visser, E., Benaissa, Z.-e.-A. and Tolmach, A.: Building Program Optimizers with Rewriting Strategies, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, ACM Press, pp. 13–26 (1998).

[54] W3C Architecture Domain: Extensible Markup Language (XML). `http://www.w3.org/XML/`.

[55] W3C Recommendation: XSL Transformations (XSLT) Version1.0 (1999). `http://www.w3.org/TR/xslt`.

[56] Yasugi, M., Komiya, T. and Yuasa, T.: Dynamic Load Balancing by Using Nested Functions and Its High-Level Description, *IPSJ Transactions on Advanced Computing Systems*, Vol. 45, No. SIG 11(ACS 7), pp. 368–377 (2004). (in Japanese).

[57] Yasugi, M., Komiya, T. and Yuasa, T.: An Efficient Load-Balancing Framework Based on Lazy Partitioning of Sequential Programs, *Proceedings of the Workshop on New Approaches to Software Construction*, pp. 65–84 (2004).

[58] Yasugi, M., Takada, J., Tabata, Y., Komiya, T. and Yuasa, T.: Primitives for Shared Memory and Its Implementation with GCC, *IPSJ Transactions on Programming*, Vol. 43, No. SIG 1(PRO 13), pp. 118–132 (2002). (in Japanese).

[59] Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proceedings of 15th International Conference on Compiler Construction (CC2006)*, Lecture Notes in Computer Science, No. 3923, Springer-Verlag, pp. 170–184 (2006).

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

Publications

Refereed Articles

1. Tasuku Hiraishi, Xiaolu Li, Masahiro Yasugi, Seiji Umatani and Taiichi Yuasa. Language Extension by Rule-Based Transformation for S-Expression Based C Languages. *IPSJ Transactions on Programming*, Vol. 46, No. SIG1 (PRO 24), pp. 40–56, 2005 (in Japanese).

2. Tasuku Hiraishi, Masahiro Yasugi and Taiichi Yuasa. Effective Utilization of Existing C Header Files in Other Languages with Different Syntaxes. *Computer Software*, Vol. 23, No. 2, pp. 225–238, 2006 (in Japanese).

3. Tasuku Hiraishi, Masahiro Yasugi and Taiichi Yuasa. A Transformation-based Implementation of Lightweight Nested Functions. *IPSJ Digital Courier*, Vol. 2, pp. 262–279, 2006.

Refereed Proceedings Articles

1. Tasuku Hiraishi, Masahiro Yasugi and Taiichi Yuasa. Implementing S-expression Based Extended Languages in Lisp. In *Proc. of International Lisp Conference (ILC2005)*, Stanford University, CA, USA, pp. 179–188, June 2005.

2. Masahiro Yasugi, Tasuku Hiraishi and Taiichi Yuasa. Lightweight Lexical Closures for Legitimate Execution Stack Access. In *Proc. of 15th International Conference on Compiler Construction (CC2006)*, Vienna, Austria, LNCS 3923, pp. 170–184, March 2006.

3. Tasuku Hiraishi, Masahiro Yasugi and Taiichi Yuasa. Experience with SC: Transformation-based Implementation of Various Language Extensions to C. In *Proc. of International Lisp Conference (ILC2007)*, Clare College, Cambridge, U.K., pp. 103-113, April 2007.

Referred Coauthor Articles

1. Masahiro Yasugi, Tasuku Hiraishi, Takenari Shinohara and Taiichi Yuasa. L-Closures: A Language Mechanism for Implementing Efficient and Safe Programming Languages. *IPSJ Transactions on Programming*, to appear (in Japanese).

Presentations

1. Tasuku Hiraishi. Design and Implementation of S-expression-based C Language suitable for extension and transformation. *JSSST Workshop on Programming and Programming Languages (PPL2003)*, Category 3, Shizuoka Prefecture, March 2003. (in Japanese).

2. Tasuku Hiraishi, Masahiro Yasugi, Tsuneyasu Komiya and Taiichi Yuasa. Design and Implementation of S-expression-based C Language suitable for extension and transformation. In *Proc. of 20th National Convention JSSST*, [1B-4], Aichi Prefectural University, September 2003. (in Japanese).

3. Ryuta Obayashi, Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani and Taiichi Yuasa. A Preliminary Implementation of a Load-balancing Framework Based on Lazy Partitioning. In *Proc. of Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP 2005)*, PRO-3, Saga Prefecture, August 2005. (in Japanese)

4. Takuya Nagasaka, Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani and Taiichi Yuasa. Towards Implementing the Source Level Checkpointing. In *Proc. of 22th National Convention JSSST*, [3A-3], Tohoku University, September 2005. (in Japanese).