A Translation Method from Smalltalk into Interoperable C Code

Kazuki YASUMATSU

August 1996

A Dissertation submitted to
the Graduate School of Engineering Science of Osaka University
in partial fulfillment of the requirements for the degree of
Doctor of Engineering

Abstract

For the delivery of production-quality applications, pure object-oriented languages such as Smalltalk and Self have some disadvantages compared to conventional languages. First, an application written in a pure object-oriented language cannot be delivered as a stand-alone machine-code version of the executable. Second, since such an application is difficult to interoperate with other programs written in other languages, it is difficult to construct a sophisticated and practical application that needs software libraries written in other languages, such as communications, graphics, and database management systems. In a pure object-oriented language, all data are represented as objects and all computation are performed by sending messages to objects. All objects are dynamically allocated and then reclaimed by garbage collection. Furthermore, some pure object-oriented languages have activation records built into first-class objects. Therefore, it is difficult to directly execute programs written in pure object-oriented languages in conventional environments and as a result, they are usually executed by an interpreter or a virtual machine. Consequently, pure object-oriented languages are not particularly suited for constructing stand-alone applications in conventional environments and interoperating with programs written in conventional languages.

One way to make pure object-oriented languages suitable for delivering applications is to translate them into an intermediate conventional language that can be compiled into machine code and can interoperate with programs written in other languages. By translating them into such an intermediate language, it is possible to deliver a stand-alone application in conventional environments and to construct an

application written in multiple languages. However, there are some issues concerning such translation. First, it may impose some restrictions on the functionality of pure object-oriented languages since a clean translation from a pure object-oriented language to a conventional language is difficult due to a language gap. Second, for preserving the functionality of pure object-oriented languages, the generated code may be so stylized that it is neither portable nor interoperable with code written in other languages. Third, the two-stage translation into machine code may result in inefficient code.

To investigate these issues, we have designed, implemented, and evaluated SPiCE that is a system for translating Smalltalk into C. Smalltalk is a pure object-oriented language and is not well suited for delivering applications because Smalltalk programs can neither run in isolation from the Smalltalk environment nor work with programs written in other languages. On the other hand, C is a portable language that is compiled into machine code and most other languages can call C procedures and use its data structures. There are two especially difficult issues that must be accounted for when translating Smalltalk code into portable, efficient, and interoperable C code while preserving the functionalities of Smalltalk. First, the execution model of Smalltalk is very different from that of C. Activation records of Smalltalk are created as first-class objects enabling complex control such as full upward funargs and exception handling, while those of C are managed as frames on a stack making such complex control difficult. Second, Smalltalk and C have very different approaches to storage management. Smalltalk has automatic storage management (garbage collection), while storage management of C is not automatic and C does not offer any support for garbage collection.

Our approach to the translation is: (1) to create runtime replacement classes implementing the same functionality of Smalltalk classes that are inherently part of the Smalltalk execution model, and (2) to provide garbage collection that is suitable for object-oriented applications and combining programs written in multiple languages, some of which do not offer any support for garbage collection. The creation of runtime

replacement classes is based on the concept of mapping activation record objects of Smalltalk onto stack frames, and mapping compiled code of Smalltalk onto machine code. The key idea of our garbage collection is the use of conservative stack scanning and indirect referencing together. Our approach fills the gaps of the execution model and the data model (storage management) in a safe and efficient manner.

Through the evaluation of SPiCE, our approach has proven to be able to generate portable, efficient, and interoperable C code, and to impose minimal restrictions on Smalltalk which enables the translation of existing Smalltalk applications as they are. For example, five large and practical Smalltalk applications, including one commercial application, have been translated without modifying the application code. Moreover, by using SPiCE, a practical application written in Smalltalk and a persistent C++ has been developed. The performance of the generated C code is roughly the same as the fastest Smalltalk implementation.

Acknowledgments

During this work, I have been fortunate to have received assistance from many individuals. First, and foremost, I would like to thank Professor Norihisa Doi of Keio University for his continuous support, encouragement and guidance of this work. He has worked very closely with me on the SPiCE project, and suggested many improvements.

I would especially like to thank Professor Katsuro Inoue who is my supervisor of this thesis and was also my advisor when I was at Osaka University, and Professor Koji Torii of Nara Institute of Science and Technology, who was my supervisor when I was at Osaka University. They have taught me invaluable skills.

I am also very grateful to the members of my thesis review committee: Professor Masaru Sudo, Professor Nobuki Tokura, Professor Tohru Kikuno, and Professor Koji Torii for their invaluable comments and helpful suggestions concerning this thesis.

I wish to express my deep thanks to Dr. Noriyuki Kamibayashi of Fuji Xerox Co., Ltd. for his valuable advice, and Takemi Yamazaki of Fuji Xerox Co., Ltd. for his advice and support of the SPiCE project.

I also wish to thank Satoshi Kurihara of NTT Corporation and Mikio Inari of Andersen Consulting for their assistance in the prototyping of SPiCE.

Thanks are also due to many Smalltalkers of Fuji Xerox Co., Ltd. who gave me many useful suggestions.

List of Major Publications

Journals and International Conferences

(1) S. Kurihara, M. Inari, N. Doi, K. Yasumatsu, and T. Yamazaki: "SPiCE Collector: The Run-Time Garbage Collector for Smalltalk-80 Programs Translated into C," in *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.

(2) K. Yasumatsu, S. Kurihara, and N. Doi: "Generation Scavenging with Ambiguous Roots," *Computer Software*, Vol. 9, No. 5, pp. 15–28, September 1993. (in Japanese).

(3) K. Yasumatsu and N. Doi: "SPiCE: A System for Translating Smalltalk Programs Into a C Environment," *IEEE Transactions on Software Engineering*, Vol. 21, No. 11, pp. 902–912, November 1995.

Other Publications

(1) K. Yasumatsu, T. Yamazaki, S. Kurihara, M. Inari, and N. Doi: "SPiCE: A System for Translating Smalltalk-80 Programs into C Environment," in *7th Conference Proceedings of Japan Society for Software Science and Technology*, pp. 325–328, October 1990. (in Japanese).

(2) S. Kurihara, M. Inari, N. Doi, K. Yasumatsu, and T. Yamazaki: "A Garbage

Collector for C Based Object-Oriented Languages," in *7th Conference Proceedings of Japan Society for Software Science and Technology*, pp. 329–332, October 1990. (in Japanese).

(3) K. Yasumatsu: "SPiCE: A System for Translating Smalltalk Programs into C Environment," *Fuji Xerox Technical Report*, No. 7, pp. 176–184, 1992. (in Japanese).

(4) K. Chiba and K. Yasumatsu: "Document Architecture Conversion for Montage V1," RSTY-TR-'93-023, Fuji Xerox, October 1993. (in Japanese).

(5) K. Yasumatsu, M. Kyojima, T. Ando, and K. Chiba: "Investigation of Document Architectures for Montage V1," RSTY-TR-'93-032, Fuji Xerox, October 1993. (in Japanese).

(6) H. Nakatsuyama, M. Kyojima, Y. Okumura, K. Yasumatsu, T. Ando, G. Uchida, K. Chiba, K. Numata, and N. Kamibayashi: "An Overview of Xebec Document Database Management System," in *IPSJ SIG Notes*, Vol. 95 of *Database System*, pp. 97–104, January 1995. 95-DBS-101. (in Japanese).

(7) M. Kyojima and K. Yasumatsu: "The Logical Structure Translation for Xebec Document Database Management System," in *IEICE Technical Report*, Vol. 94 of *Office System*, pp. 13–18, March 1995. OFS 94-53. (in Japanese).

Patents

(1) K. Yasumatsu and T. Satoh: "File name length augmentation method," U.S. Patent, No. 5307494, April 1994.

Free Softwares

(1) K. Yasumatsu: "SmallWalker: A Smalltalk World-Wide Web Browser," September 1995, Available for anonymous internet FTP from st.cs.uiuc.edu in

/pub/Smalltalk/st80_r41/SmallWalker1.0/.

This software was used for CS497 class project at the University of Illinois.

Contents

1 Introduction **1**

1.1 Application Delivery of Object-Oriented Languages 1

1.2 Translation Approach to Application Delivery 3

1.3 Outline of the Thesis 4

2 Background and Related Work **7**

2.1 Smalltalk 7

2.1.1 The Smalltalk Language 7

2.1.2 The Smalltalk System 10

2.2 Related Work 11

2.2.1 Producer 11

2.2.2 Smalltalk Application Compiler 11

2.2.3 Orchard 12

2.2.4 Babel 12

2.2.5 Smalltalk/X 13

2.2.6 Summary of Related Work 13

3 SPiCE: A System for Translating Smalltalk Programs into a C Environment **14**

3.1 Background 14

3.2 Difficulties of the Translation 15

3.3 Key Concept of the Translation 17

3.4 Overview of the Translation 18
3.5 Translating Smalltalk into C 20
3.5.1 Primitive Builtin Types 20
3.5.2 Literals 20
3.5.3 Variables 21
3.5.4 Messages 23
3.6 Runtime Replacement Classes 24
3.6.1 Classes 24
3.6.2 Processes 24
3.6.3 Blocks 25
3.6.4 Exception Handling 26
3.7 Garbage Collection 29
3.8 Application Booting 31
3.9 Summary 31

4 Generation Scavenging with Ambiguous Roots **33**

4.1 Requirements for Garbage Collection 34
4.2 Conservatism in Garbage Collection 34
4.2.1 Conservative Collector 34
4.2.2 Partially Conservative Collector 35
4.2.3 Reference Counting Collector 36
4.3 Outline of Generation Scavenging with Ambiguous Roots 37
4.3.1 Generation Scavenging 37
4.3.2 Conservative OOP-Finding 39
4.4 Collector I 41
4.5 Dividing Object Table 43
4.6 Collector II 44
4.7 SPiCE Collector 47
4.7.1 Object Formats 47
4.7.2 Intergenerational References 49

4.7.3 Roots of the System 50
4.7.4 Tenuring Policy 50
4.8 Summary 53

5 Evaluation of SPiCE 54
5.1 Restrictions on Smalltalk 55
5.2 Interoperability with C 57
5.3 Performance in C 59
5.4 Performance of Garbage Collection 64
5.5 Comparison with Related Work 67

6 Conclusion 68
6.1 Summary of Main Results 68
6.2 Future Work 69

Bibliography 72

A Algorithm of Garbage Collection 79
A.1 Algorithm of the Collector I 79
A.2 Algorithm of the Collector II 83

List of Figures

3.1 Mapping of execution models between Smalltalk and SPiCE 18
3.2 Overview of the translation process 19
3.3 Example of the translation of literals 21
3.4 Example of the translation of variables 22
3.5 Comparison of processes in Smalltalk and SPiCE 25
3.6 Snapshots of a block evaluation in Smalltalk and SPiCE 27
3.7 Example of the translation of blocks 28

4.1 Heap structure of generation scavenging 38
4.2 Conservative OOP-finding 40
4.3 Heap structure of the collector I 42
4.4 Heap structure of the collector II 46
4.5 Object formats of SPiCE 48
4.6 Memory layout 51

5.1 SmallWalker: A Smalltalk WWW browser 56
5.2 Combination of Smalltalk and C 58
5.3 Xebec document class browser 60

List of Tables

2.1 Summary of related work 13

5.1 Smalltalk system benchmarks 62

5.2 Benchmark running times 63

5.3 SPiCE runtime profile 64

5.4 Performance of the SPiCE collector 65

5.5 Runtime overhead of garbage collection 66

Chapter 1

Introduction

1.1 Application Delivery of Object-Oriented Languages

For the delivery of production-quality applications, pure object-oriented languages such as Smalltalk [Goldberg and Robson 1983] and Self [Ungar and Smith 1987] have some disadvantages compared to conventional languages. First, an application written in a pure object-oriented language cannot be delivered as a stand-alone machine-code version of the executable. The application can be delivered as source code or as an interpreted-code version of the executable with an interpreter. Delivering source code does not allow for confidentiality. Delivering an interpreted-code version of the executable with an interpreter often requires the user to purchase, in addition to the application, a license for the interpreter. For these reasons, a stand-alone executable is the most desirable when delivering an application. Second, a program written in a pure object-oriented language is difficult to interoperate with other programs written in other languages. From the viewpoint of the construction of production-quality applications, the ability of two or more programs written in different programming languages to work together, or language interoperability, is important. The need for interoperability arises in many contexts. Generally, the desire to combine programs

written in different languages springs from the availability of specific capabilities in some particular language or existing program. If a language lacks interoperability, it is difficult to develop a sophisticated and practical application that needs software libraries written in other languages, such as communications, graphics, and database management systems.

On the other hand, hybrid object-oriented languages such as C++ [Stroustrup 1986] and Objective-C [Cox 1986] are suited for delivering applications. They are grafted onto some other, usually conventional, languages and they allow programmers to intermix a lower level language and remove layers of abstraction. A program written in a hybrid object-oriented language can be compiled into a stand-alone executable directly or indirectly through a lower level language, and works with other programs written in a lower level language or other languages through the lower level language.

However, in a pure object-oriented language, all data, including low-level basic data such as integers, characters, and arrays, are represented as objects. Furthermore, all computation, including low-level operations such as variable accessing, arithmetic, and array indexing, are performed by sending messages to objects. All objects are dynamically allocated and then reclaimed by garbage collection. Furthermore, some pure object-oriented languages have activation records built into first-class objects enabling complex control such as full upward funargs and exception handling. Therefore, it is difficult to directly execute programs written in pure object-oriented languages in conventional environments and as a result, they are usually executed by an interpreter or a *virtual machine*. Consequently, pure object-oriented languages are not particularly suited for constructing stand-alone applications in conventional environments and interoperating with programs written in conventional languages.

1.2 Translation Approach to Application Delivery

There are two approaches for producing a stand-alone executable for pure object-oriented languages: compiling the application into machine code directly, or translating the application into source code for another language and then compiling the translated code. For language interoperability, the latter approach is preferable.

Most existing approaches for supporting interoperability have been based on the use of the *remote procedure call* (RPC) [Birrell and Nelson 1984] for coordinating the execution of the interoperating programs [Sun 1985; Jones *et al.* 1985; Bershad *et al.* 1987; Gibbons 1987]. The problem of the RPC-based approach is the lack of "closely coupled" interoperation between programs written in different languages. Closely coupled means that an application which is as real-time or sophisticated as a device driver or a database management system might have different parts written in different languages [Weiser *et al.* 1989]. The parts could share data structures, an address space, and threads of control. RPC, even when local but across address spaces, is usually much more expensive than calls within the same address space.

Another approach for supporting interoperability is the use of a common base language to which other languages must conform. Languages must be able to intercall procedures with the common base language and to mix their data structures with those of the common base language, or languages must be translated into the common base language. This approach meets the closely coupled interoperation.

By using an intermediate language as a common base language, translating the application into source code for such an intermediate language is suitable not only for the construction of stand-alone executables but for language interoperability. Translating one language to another is a very old idea. *Universal Computer Oriented Language* (UNCOL) [Strong *et al.* 1958] was first documented in 1958. UNCOL is an intermediate language for all high-level languages and it is compiled for a specific

architecture. The benefits of UNCOL are: (1) UNCOL provides a common meeting point for all languages and thus enables the combination of programs written in multiple languages; (2) by emitting UNCOL rather than machine codes, a compiler is extremely portable; (3) the UNCOL compiler can simplify the individual language compilers by providing language-independent optimization.

Today, the C language [Kernighan and Ritchie 1978] could be considered as a possible UNCOL. Most operating systems have a C compiler and most other languages can call C procedures and use its data structures. The C compiler provides some level of machine-specific optimization. Also, many valuable software libraries and systems written in C are available.

By using a translation approach for pure object-oriented languages, that is, translating them into an intermediate language like C, it is possible to deliver a stand-alone application in conventional environments and to construct an application written in multiple languages. However, there are some issues on such translation:

1. It may impose some restrictions on the functionality of pure object-oriented languages since a clean translation from a pure object-oriented language to a conventional language is difficult due to a language gap.

2. For preserving the functionality of pure object-oriented languages, the generated code may be so stylized that it is neither portable nor interoperable with code written in other languages.

3. The two-stage translation into machine code may result in inefficient code.

The first two are antagonistic issues against each other, and they become more problematic when accounting for both of them at the same time.

1.3 Outline of the Thesis

To investigate these issues, we have designed, implemented, and evaluated SPiCE that is a system for translating Smalltalk into C. Smalltalk is a pure object-oriented

language and is not well suited for delivering applications because Smalltalk programs can neither run in isolation from the Smalltalk environment nor work with other programs written in other languages. On the other hand, C is a portable language and can be considered as a possible UNCOL, i.e., the C compiler produces efficient machine code and most other languages can call C procedures and use its data structures.

There are two especially difficult issues concerning the translation of Smalltalk code into portable, efficient, and interoperable C code while preserving the functionalities of Smalltalk. First, the execution model of Smalltalk is very different from that of C. Activation records of Smalltalk are created as first-class objects enabling complex control such as full upward funargs and exception handling, while those of C are managed as frames on a stack making such complex control difficult. Second, Smalltalk and C have very different approaches to storage management. Smalltalk has automatic storage management (garbage collection), while storage management of C is not automatic and C does not offer any support for garbage collection. Because of these difficulties, while many work on translating Smalltalk into an intermediate language has been done [Cox and Schmucker 1987; Vokach-Brodsky and Wolczko 1990; Nash and Haebich 1991; Moore *et al.* 1994; Cla 1994], no one has yet been able to generate portable, efficient, and interoperable intermediate code while preserving the functionalities of Smalltalk.

First, this thesis proposes the method used in SPiCE for translating Smalltalk into C. The key feature of the translation is the creation of runtime replacement classes implementing the same functionality of Smalltalk classes that are inherently part of the Smalltalk execution model. The creation of runtime replacement classes is based on the concept of mapping activation record objects of Smalltalk onto stack frames, and mapping compiled code of Smalltalk onto machine code. The runtime replacement classes encapsulate the differences of the execution model between Smalltalk and C, and enables the generation of portable, efficient, and interoperable C code while preserving the functionalities of Smalltalk.

We also propose a new garbage collection technique that is suitable for object-oriented applications and combining programs written in multiple languages, some of which do not offer any support for garbage collection. The key idea of our garbage collection is the use of conservative stack scanning and indirect referencing together. This technique fills the gaps of the data model (storage management) between Smalltalk and C, and enables the generated C data structures (Smalltalk objects) to be mixed with other languages' data structures.

Then, we show the effectiveness of the translation method and the garbage collection technique by presenting an evaluation performed with practical applications. We also show the usefulness of the translation approach to interoperability by presenting a practical application written in multiple languages.

In Chapter 2, we give some background used through the thesis, and review related work. Chapter 3 presents the translation method from Smalltalk into C through the design and implementation of SPiCE. Chapter 4 proposes a garbage collection technique used in SPiCE. Chapter 5 presents an experience developing a practical application written in Smalltalk and a persistent C++, and shows the performance of the generated C code by SPiCE and the garbage collection of SPiCE. Finally, Chapter 6 summarizes the main results of the thesis and presents some areas for future research.

Chapter 2

Background and Related Work

In this chapter, we present background materials. First, Section 2.1 gives a brief introduction to the Smalltalk language and programming environment. Then, Section 2.2 reviews related work.

2.1 Smalltalk

Smalltalk is more than a language. It is the combination of a language, a set of reusable objects, a programming environment, and a *virtual machine*. In the following two subsections, we review some of the more unique features of the Smalltalk system.

2.1.1 The Smalltalk Language

The Smalltalk language [Goldberg and Robson 1983] is a pure object-oriented language. All data are objects, all expressions are message sends, and all computation is performed by sending messages to objects. Each object is an instance of a *class* that is itself an object.

Smalltalk programs contain no static type declarations. Instead, type-correctness is verified during execution, i.e., Smalltalk is dynamically typed. Dynamic typing maximizes expressiveness by eliminating the need for more restrictive static type

checking. It is an important factor in attaining the highest possible degree of polymorphism and therefore code reuse. Message sends are always dynamically dispatched: the *method* (procedure) invoked depends on the dynamically computed receiver of the message.

Unlike many earlier Lisps which used dynamically scoped variables, Smalltalk's variables are lexically scoped. As a result, the variable bindings in methods can be determined at compile time.

Smalltalk has only a small set of basic control structures built in: message sends, *blocks* (lexical closures), non-local return, and a few more. Other control structures such as conditional statements and while-loops are constructed out of the basic ones. For example, in the following statements, a while-loop is constructed by the `whileTrue:` message to a block object, and a conditional statement is constructed by the `ifTrue:` message to a boolean object:

```
| string index |
string := 'abcDefg'.
index := 1.
[index <= string size]
    whileTrue:
        [(string at: index) isUppercase
            ifTrue: [^'Has an uppercase character'].
        index := index + 1].
^'Not have an uppercase character'.
```

In the above statements, the Smalltalk compiler can eliminate those message sends and thus can generate efficient code. However, the Smalltalk compiler cannot optimize the following statements:

```
| string index block1 |
string := 'abcDefg'.
index := 1.
block1 :=
    [| block2 |
    block2 := [^'Has an uppercase character'].
    (string at: index) isUppercase ifTrue: block2.
    index := index + 1].
[index <= string size]
    whileTrue: block1.
^'Not have an uppercase character'.
```

In these statements, `block1` and `block2` are lazy-evaluated and a non-local return occurs in the evaluation of `block2`.

As described above, a block is not a syntactic sugar. A block is an object and it is lazy-evaluated by a `value` message. A block can refer to values of temporary variables even after control has returned from the method the block was created in. This is the so-called *upward funarg problem.* The method return (^-return) from a block is a non-local return, and returns to the caller of the method the block was created in.

Smalltalk has the multiprocessing facility in the form of process and semaphore objects. A process object represents an independent thread of control and a semaphore object provides synchronized communication between processes. Smalltalk processes run in the same address space and can be thought of as *lightweight processes* (or *threads*).

Smalltalk also has the exception handling facility in the form of signal and exception objects. A normal case and an exceptional case are represented as different blocks, and they are passed as arguments of a `handle:do:` message to a signal object. When an exception occurs, an exception object is created, and it searches the prepared block for the exceptional case (or *exception block*) through the call chain, and evaluates the exception block with itself as an argument. By sending a message to the exception object in the exception block, the thread of control is continued, returned, or restarted. The last two cases cause the call chain to be unwound to the

`handle:do:` message invocation.

2.1.2 The Smalltalk System

The Smalltalk system contains a large number of reusable objects. These objects implement general data structures such as integers, strings, sets, dictionaries, and points. Other objects implement an interactive programming environment, including text editors, browsers, and debuggers [Goldberg 1984].

The Smalltalk system is built around the concept of a *virtual image*. The virtual image is a dynamic data structure representing all code and data in the Smalltalk system. Applications are built by incrementally adding, modifying, or reusing such code and data. As a result, productivity with the Smalltalk environment is much higher than that with conventional environments. However, applications cannot run in isolation from the Smalltalk environment because they are part of and dependent upon the Smalltalk environment.

The lower layer of the virtual image is the *virtual machine*. The Smalltalk compiler generates code for the virtual machine, and the virtual machine interprets this code. Two special features of the virtual machine are the creation of *contexts* (activation records) as first-class objects, and automatic storage reclamation (garbage collection). Contexts are first-class objects that allow reflecting upon the activation records of the Smalltalk system. For example, the exception handling facility of Smalltalk is written in Smalltalk itself by an explicit use of contexts. Contexts are inherently part of the virtual machine's architecture, or the execution model of Smalltalk. Furthermore, contexts are reclaimed by garbage collection instead of being managed in an LIFO stack.

Because of these two features, it is difficult to share threads of control and data structures with other languages, that is, interoperation with programs written in other languages is difficult.

2.2 Related Work

Since Smalltalk is one of the earliest object-oriented languages, many work on translating Smalltalk into an intermediate language has been done. We now review related work.

2.2.1 Producer

Producer [Cox and Schmucker 1987] is a tool for translating Smalltalk into Objective-C, and through the use of the Objective-C compiler, into C. Its purpose is to integrate the strengths of production programming environments like C/UNIX with rapid prototyping environments like Smalltalk into a comprehensive hybrid environment. The code generation phase of Producer is divided into two passes. The first pass determines the types of objects by using typing information, and the second generates Objective-C code. Typing information is provided in the form of a database of rules. The programmer is required to supply application-specific rules that provide type information for the classes that make up the application. Producer does not support functionalities such as blocks and garbage collection since Objective-C does not have them. Therefore, it needs help from the programmer for guiding the translation or modifying programs so that they can be translated. It is not a general translator and it imposes much restrictions on Smalltalk. Unfortunately, it was unable to measure the performance of the translated application because a complete set of rules had not yet been implemented for the translator.

2.2.2 Smalltalk Application Compiler

Smalltalk application compiler [Yokach-Brodsky and Wolczko 1990] is designed to translate Smalltalk into object-oriented compiler languages such as C++ or Eiffel. Its purpose is to produce a stand-alone machine-code version of the Smalltalk application. It proposes a general mapping of the object models between Smalltalk and other object-oriented compiler languages. The functionalities that are not supported by

the target language. such as blocks and garbage collection, are written in the target language or rely on the functionalities of the target language. It was not completely implemented.

It does not identify nor inference the types of objects, and therefore it translates the classes describing basic types such as integer or character to the target language. This approach to types is also adopted by the following three translators.

2.2.3 Orchard

Orchard [Nash and Haebich 1991] is a tool to translate a particular Smalltalk application into C. Its purpose is to separate Smalltalk applications from the Smalltalk environment. A unique feature of it is a creation of *standard replacement classes* that implement the functionality of the Smalltalk standard classes such as integers or characters in a different manner. The application to be translated by Orchard must use the standard replacement classes instead of the original Smalltalk classes. It supports garbage collection but does not support blocks. It is not a full translator and imposes much restrictions on Smalltalk. Applications translated by Orchard run somewhere between a similar speed to 1/3 the speed of the original Smalltalk.

2.2.4 Babel

Babel [Moore *et al.* 1994] is a translator from Smalltalk into *Common Lisp Object System* (CLOS) [Bobrow *et al.* 1988; Keene 1989]. Its purpose is to produce a stand-alone machine-code version of the Smalltalk application. It imposes minimal restrictions on Smalltalk since CLOS has rich functionalities such as message sending, lexical-closures, non-local return, and garbage collection. However, Lisp implementation is not usually suited for producing a stand-alone machine-code version of the executable and interoperating with programs written in conventional languages. Applications translated by Babel are between 4.5 times to 10 times slower than the original Smalltalk.

2.2.5 Smalltalk/X

Smalltalk/X [Cla 1994] is the combination of a Smalltalk to C compiler and a runtime system. An application of Smalltalk/X is compiled into a stand-alone executable. Smalltalk/X supports almost all functionalities of Smalltalk such as blocks. processes. exception handling. and garbage collection, and thus it imposes minimal restrictions on Smalltalk. However. its compiler generates neither portable nor interoperable C code. The runtime system creates contexts as in the Smalltalk virtual machine and the generated C code is assumed to be executed by using the contexts. As a result, the generated C code does not follow the C procedure call/return mechanism and therefore inter-calling procedures with other C programs is not straightforward. Furthermore. programmers must be careful of mixing data structures. A Smalltalk/X object must be referred to from the contexts so as not to be reclaimed by garbage collection. If an object is referred to only from a native C stack. i.e.. local C variables. it is reclaimed by garbage collection and this results in a *dangling pointer*.

2.2.6 Summary of Related Work

We characterize the related work with respect to three issues on the translation described in Section 1.2: restrictions on Smalltalk; the generation of interoperable code: and the generation of efficient code. Table 2.1 gives a summary of the related work. No one has yet been able to generate interoperable and efficient intermediate code while preserving the functionalities of Smalltalk.

Table 2.1: Summary of related work

	Restrictions	Interoperability	Efficiency
Producer	Much	Good	No data
Application Compiler	Much	Good	No data
Orchard	Much	Good	1 time to 3 times slower
Babel	Minimal	Not Good	4.5 times to 10 times slower
Smalltalk/X	Minimal	Not Good	No data

Chapter 3

SPiCE: A System for Translating Smalltalk Programs into a C Environment

We present SPiCE that is a system for translating Smalltalk programs into a C environment. In this chapter, it is assumed that the reader has some familiarity with C. First, Section 3.1 gives a background of SPiCE, and Section 3.2 describes difficulties of the translation. Then, Section 3.3 presents the key concept of the translation, and Section 3.4 gives an overview of the translation. Following this, Section 3.5 describes how Smalltalk can be translated into C, and Section 3.6 describes runtime replacement classes. Section 3.7 gives an overview of the garbage collection of SPiCE. Section 3.8 discusses how to boot the translated applications. Finally, Section 3.9 summarizes SPiCE.

3.1 Background

Smalltalk is very well suited for prototyping of applications but it is less well suited for delivering applications because Smalltalk programs can neither run in isolation from the Smalltalk environment nor work with other programs written in other languages.

One way to make Smalltalk suitable for delivering applications is to translate Smalltalk into a conventional language such as C. C is a portable language that is compiled into machine code and most other languages can call C procedures and use its data structures. By translating Smalltalk code into portable and interoperable C code, it is possible to deliver a stand-alone machine-code version of the Smalltalk application, and to construct an application written in Smalltalk, C, and other languages through C.

We have designed and implemented SPiCE that is a system for translating Smalltalk into C. The goals of the SPiCE project are:

1. To make possible the delivery of a stand-alone machine-code version of the Smalltalk application.

2. To translate existing practical Smalltalk applications as they are. This means that Smalltalk functionalities used by such applications, such as processes, blocks, exception handling, and garbage collection, must be preserved.

3. To improve C interoperability of Smalltalk by generating interoperable C code. This enables the construction of an application written in Smalltalk, C, and other languages through C.

4. To execute the translated application faster than the original Smalltalk.

Last three goals are linked to three issues on the translation described in Section 1.2.

3.2 Difficulties of the Translation

There are two especially difficult issues concerning the translation of Smalltalk code into portable, efficient, and interoperable C code while preserving the functionalities of Smalltalk. First, the execution model of Smalltalk is very different from that of

C. Activation records of Smalltalk are created as first-class objects enabling complex control such as full upward funargs and exception handling, while those of C are managed as frames on a stack making such complex control difficult. Second, Smalltalk and C have very different approaches to storage management. Smalltalk has automatic storage management (garbage collection), while storage management of C is not automatic and C does not offer any support for garbage collection.

One approach for preserving the functionalities of Smalltalk is to provide a runtime system that supports the facilities of the virtual machine such as contexts, and to generate C code that is executed by using the facilities provided by the runtime system. This approach is used in *Smalltalk/X* [Cla 1994]. A similar approach is used in *Kyoto Common Lisp* (KCL) [Yuasa and Hagiya 1985]. KCL is a Common Lisp system, and its Lisp compiler generates C code instead of generating machine code directly. The generated C code is assumed to be executed by using the KCL interpreter stacks that enable special control and garbage collection. However, this approach fails to generate interoperable C code. The generated C code does not follow the C procedure call/return mechanism and therefore inter-calling procedures with other C programs is not straightforward. Furthermore, programmers must be careful of mixing data structures. The generated C data structure (or an object) must be referred to from a context or a special stack so as not to be reclaimed by garbage collection. If an object is referred to only from a native C stack, i.e., local C variables, it is reclaimed by garbage collection and this results in a *dangling pointer*.

By interoperable C code, we mean that the generated C code must be easily and efficiently combined with other C programs or code written in other languages, and the generated C data structures must be safely mixed with other languages' data structures. The generated C code should use as much of C facilities as possible: the generated C code should follow the C procedure call/return mechanism, and the generated C data structures should be safely referred to from C variables. We do not want a context or a special stack to enable complex control, or a tagged pointer to enable garbage collection, as this is inefficient and undesirable for interoperability.

3.3 Key Concept of the Translation

As described in Section 2.1.2, the virtual image includes all code and data in the Smalltalk system. All code and data are represented as objects, and each object is an instance of a class that is itself an object. Translating Smalltalk into C means representing the code and data of Smalltalk in C.

However, there are some classes that are inherently part of the virtual machine's architecture. Their functionality, which includes processes, exception handling, and blocks (*lexical closures*), is used at the application level. These classes cannot be represented in C on the micro processor's architecture, because the virtual machine creates contexts as first-class objects and instances of these classes refer to contexts. While activation records of C are managed as frames on an LIFO stack, contexts are created as objects and some situations produce non-LIFO control.

To solve this problem, we created *runtime replacement classes* that implement the same functionality of such classes on the micro processor's architecture. The implementation of the runtime replacement classes is based on the concept of mapping contexts onto stack frames, and mapping compiled code (*compiled methods*) onto machine code. The lower layer of the runtime replacement classes is a runtime system that supports their functionalities. The runtime system also supports method lookup, primitive operations, and garbage collection. An instance of the runtime replacement classes can be used the same as in the Smalltalk system but it includes pointers to stack frames or machine code that are interpreted and managed only by the runtime system. The runtime replacement classes encapsulate the differences of the execution model between Smalltalk and C, without referring to implementation details by preventing an object from being manipulated except via its defined external operations. Fig. 3.1 shows a mapping of the execution models between Smalltalk and SPiCE.

Another problem is that objects created by the generated C code by SPiCE must be reclaimed when they are no longer referred to. To solve this problem, we have designed a garbage collection technique, called *generation scavenging with ambiguous roots*, that is suitable for object-oriented applications and combining programs

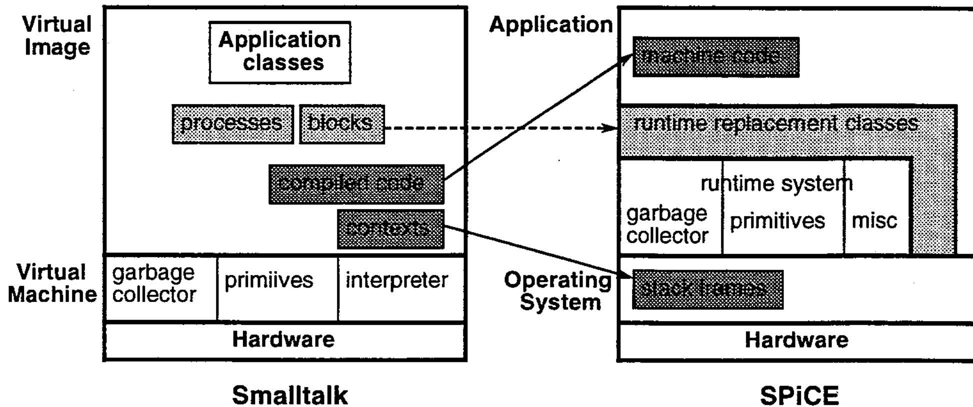


Figure 3.1: Mapping of execution models between Smalltalk and SPiCE

written in multiple languages, some of which do not offer any support for garbage collection. The key idea of our garbage collection is the use of conservative stack scanning and indirect referencing together. Conservative stack scanning enables the generated C data structures (Smalltalk objects) to be mixed with other languages' data structures. Indirect referencing enables the use of generational copying garbage collection algorithm in a conservative manner. An overview of our garbage collection technique is described in Section 3.7.

3.4 Overview of the Translation

The translator of SPiCE is written in Smalltalk, which provides the following advantages. First, an existing Smalltalk compiler can be utilized, and second, static analysis of the Smalltalk text, such as type inferencing, becomes easier due to the information from the Smalltalk environment. Its runtime replacement classes and runtime system is written in C. The runtime replacement classes were first translated

into from the original classes by the translator, then were modified by hand.

A method (procedure) is realized as a C function, a class definition as a definition of the structure of its instance, and an object as data of the type given by its class. A class object is also realized as data of the type given by its class (*metaclass*). When the translator is invoked to translate a class, it generates C functions corresponding to instance methods and class methods of the class, and C data corresponding to a class object, a metaclass object, class variable objects, and literal objects. To translate objects shared with several classes such as shared variable objects, the translator can translate a specified object into C data. Fig. 3.2 shows an overview of the translation process in SPiCE. Classes and shared objects required by an application are translated by the translator and the generated C code becomes executable when linked with the runtime replacement classes and the runtime system. Other code originally written in C also can be linked.

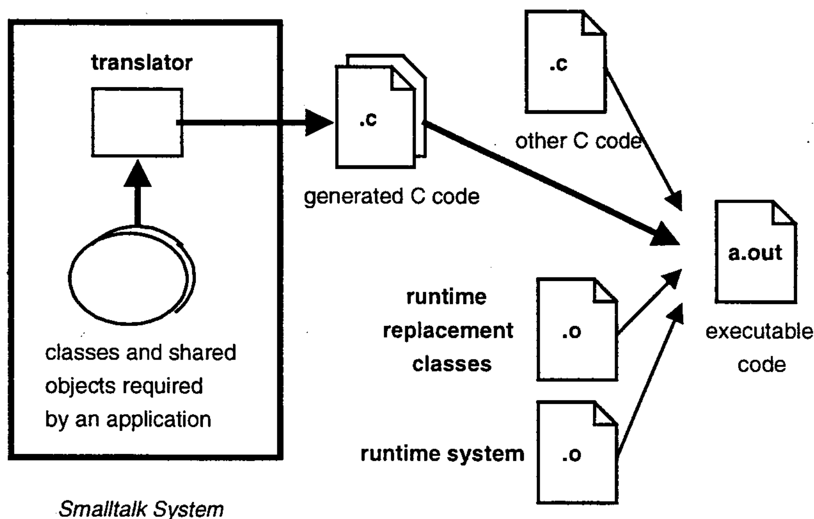


Figure 3.2: Overview of the translation process

3.5 Translating Smalltalk into C

3.5.1 Primitive Builtin Types

The C language defines the basic data types, such as `int`[1], `float`, and `char` and the basic operations for these data types. In Smalltalk, corresponding data types are defined as classes, such as SmallInteger[2], Float, and Character. Instances of these classes can respond to messages and a programmer can modify their operations. Mapping these classes to the basic types of C improves the efficiency of the generated code. However, we define these classes as normal classes to preserve the semantics of Smalltalk. Another reason for not mapping them is that type inferencing of Smalltalk is very difficult because Smalltalk has no type (or it is a dynamically-typed language).

3.5.2 Literals

Smalltalk has six types of literals that refer to constant objects. The six types are: numbers, characters, strings, byte arrays, symbols, and arrays of other literals. Literals for SmallInteger and Character, which are immediate objects[3], are translated into bit operations that convert integers or characters of C to tagged representations[4] of SmallInteger or Character. The other literals are translated into pointers to static data in the functions corresponding to the methods. However, literals for symbols are translated into pointers to global data so that they are unique in the translated application program.

Fig. 3.3 shows an example of the translation of literals. The prefix '`sp_`' is used for names, so that similar names do not interfere with other C programs. The type `sp_otEntry` represents an object table entry. We will describe the object table in more details in section 3.7.

---

[1] Note that a typewriter font is used for C code fragments.

[2] Note that a sans serif font is used for Smalltalk code fragments.

[3] Immediate objects are stored within a machine word as tagged immediate values.

[4] These tagged representations are used to identify a class of an immediate object, not to realize garbage collection.

```
/*
The original Smalltalk code is:
'a string'. 10. #symbol.
*/
#define sp_asSmallInt(n) ((n)<<shiftX | tagX)
typedef struct { word_t *dataPtr; ... } sp_otEntry, *sp_oop;

static char sp_literal1_data[] = "a string";
static sp_otEntry sp_literal1 = { (word_t*)sp_literal1_data, ... };
extern sp_otEntry s_symbol;

&sp_literal1; sp_asSmallInt(10); &s_symbol;
```

Figure 3.3: Example of the translation of literals

A block is also thought to be a literal. The translation of blocks is not straightforward. A block literal refers to an instance of BlockClosure and this instance is lazy-evaluated by a value message. Since an instance of BlockClosure refers to a compiled method and a context. BlockClosure is a runtime replacement class. Blocks are described in Section 3.6.3.

3.5.3 Variables

Smalltalk has the following four types of variables: temporary variables. instance variables. shared variables. and pseudo variables. Temporary variables are private within a method and are translated into temporary variables in the C function. The receiver and arguments of a method are passed as arguments of the C function. The instance variables of a receiver object are translated into member access to the corresponding C structure. In Smalltalk. shared variables. such as global variables. pool variables. and class variables. are Association objects in Dictionary objects that represents name scopes and this structure is visible to the programmer. For example, an expression 'Smalltalk includesKey: #Association' is used to test if a class named Association exists in the system. To preserve this semantics. shared variables and the

name scope dictionaries are translated into global data. Pseudo variables such as true, false and nil are translated into pointers to the corresponding global data. Pseudo variable thisContext, which refers to an active context, is not supported. This pseudo variable is only used in classes that are inherently part of the Smalltalk execution model and these classes are supported as the runtime replacement classes.

Fig. 3.4 shows an example of the translation of variables. Macro 'P2' is used for arranging the order of the arguments and this is described in the next section.

```
/*
The original Smalltalk code is:
(Association)example: arg
    | tmp |
    tmp := arg.
    tmp := key. "instance variable"
    tmp := Association. "global variable"
    tmp := nil.
*/
typedef struct { sp_oop key, value; } sp_Association;
extern sp_otEntry sp_g_Association, sp_nil;
#define sp_sval(anAsc)  ((sp_Association*)(anAsc).dataPtr)->value
#define nil             (&sp_nil)

sp_oop sp_m_Association_example_(P2(self, arg))
sp_oop self, arg;
{
    sp_oop tmp;

    tmp = arg;
    tmp = ((sp_Association*)self->dataPtr)->key;
    tmp = sp_sval(sp_g_Association);
    tmp = nil;
    return self;
}
```

Figure 3.4: Example of the translation of variables

3.5.4 Messages

A message expression is translated into a runtime routine call that looks up and calls the function corresponding to the method that will be invoked by the message. The key of a method lookup is a symbol object that is a literal. One implementation problem is that, in Smalltalk, the receiver is evaluated first, and the arguments are evaluated in the order from left to right, but the order of evaluation of arguments in C depends on the platform (e.g. the C compiler). To solve this problem, macros with arguments are used at the runtime lookup routine call for arranging the order of arguments according to the platform.

The following four types of message expressions are translated specially:

1. A message that represents a control structure such as ifTrue:ifFalse: or whileTrue: is translated into the control structure of C such as `if` or `while` statements, if possible.

2. An arithmetic message such as +, −, *, or / is translated into a runtime routine call that executes an arithmetic primitive when the receiver is SmallInteger or Float. If the receiver is neither SmallInteger nor Float, or if the arithmetic primitive fails, normal method lookup is performed.

3. If the class of a message receiver can be identified at translation time, a function corresponding to the method invoked by the message is looked up and this message is translated into a call to the function. The class of a receiver can be identified from Smalltalk text only if the receiver is a literal, a block (literal), or a class name. If the receiver is the pseudo-variable super, the function invoked by the message can be statically looked up. If the receiver is the pseudo-variable self and the class in which the method is declared has no sub-classes, the function invoked by the message can also be statically looked up. We do not perform any other kind of type inferencing.

4. We selected frequently used messages, such as at:, at:put:, and size, from the Smalltalk environment. For each class, functions corresponding to the methods

invoked by these frequently used messages are looked up at translation time and these functions are registered in a table prepared in a class data structure. A frequently used message is translated into a runtime routine call that selects a function from the table in the class of the receiver and calls it.

3.6 Runtime Replacement Classes

The generated C code becomes executable when linked with the runtime replacement classes and the runtime system. The following eight classes are runtime replacement classes: Behavior, Class, Metaclass, Process, Semaphore, BlockClosure, Exception, and Signal. Smalltalk classes that support programming, such as Compiler and Debugger, are not the target of the translation. All other classes, except for CompiledMethod and Context, can be translated as they are. The runtime system manages the runtime replacement classes and supports method lookup, primitive operations, and garbage collection. In this section, we describe the runtime replacement classes. We do not describe the runtime system in detail except for garbage collection. An overview of the garbage collection of SPiCE is described in Section 3.7.

3.6.1 Classes

Behavior, Class, and Metaclass provide the Smalltalk class mechanism. They are realized as they are, except for a reference to a method dictionary. The method dictionary is realized as a table that is composed of pairs of message selectors, i.e., symbol objects and C functions. This table is used for the method lookup by the runtime system. Instances of these classes also have a table for frequently used messages as described in Section 3.5.4.

3.6.2 Processes

Process represents an independent thread of control and Semaphore provides synchronized communication between processes. Smalltalk processes run in the same address

space and each process has a chain of contexts that represents a thread of control. A Smalltalk process is thought to be *lightweight processes* (or *threads*), and a context chain can be mapped onto a C language execution stack (or *C-stack*). The runtime system supports lightweight processes and allocates C-stacks for each process object. We call these C-stacks *user stacks*. On the other hand, we call a single C-stack that is allocated by the operating system the *kernel stack*. The kernel stack is used only at the time of system initialization, system finalization, and garbage collection. During process switching, registers are saved on a user stack, its stack pointer is saved in a process object, then the user stack is switched. Fig. 3.5 shows a comparison of processes in Smalltalk and SPiCE.

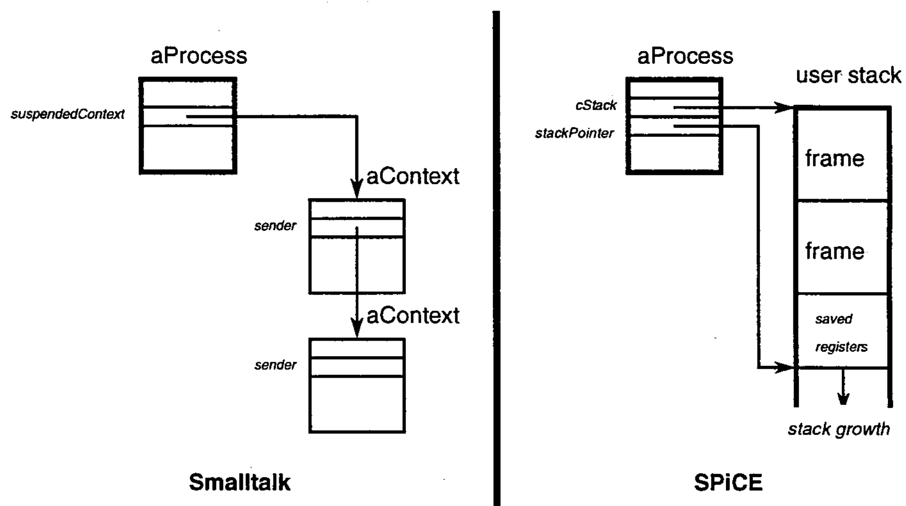


Figure 3.5: Comparison of processes in Smalltalk and SPiCE

3.6.3 Blocks

An instance of BlockClosure (a *block*) is created by a block literal and it is lazy-evaluated by a value message. A block can refer to values of temporary variables

even after control has returned from the method the block was created in. This is the so-called *upward funarg problem.* The method return (^-return) from a block is a non-local return, and returns to the caller of the method the block was created in.

A block literal is translated into a function (called a *block function*) corresponding to the text in the block and a runtime routine call that creates a block. The realized block has three instance variables. The first instance variable is used to refer to the block function. When the block is lazy-evaluated, its block function is called with the arguments: the evaluated block itself and the block arguments. The second instance variable is used to refer to an array object that is a space for values of the variables out of the block's scope. Such variables are translated so that their corresponding C code refers to the contents of the array object. This array object exists without being reclaimed by garbage collection even after control has returned from the method because it is referred to from the block. This is a simple solution to the upward funarg problem. The third instance variable is used to refer to the block that represents the outer scope and is used to chain the nested blocks.

The method return is implemented using the C library functions for non-local return (`setjmp` and `longjmp`). When a method being translated includes a method return from a block, the translator generates the `setjmp` call to save the runtime state (i.e., registers). The runtime state is saved on the user stack and is referred to from the block using the third instance variable. When a method return from the block is executed, the runtime state is searched for through the third instance variable of the blocks and `longjmp` is called with the runtime state as an argument to continue execution from the corresponding `setjmp` call. Fig. 3.6 shows snapshots of a block evaluation in Smalltalk and SPiCE, and Fig. 3.7 shows an example of the translation of blocks.

3.6.4 Exception Handling

Exception and Signal provide the exception handling facility. A normal case and an exceptional case are represented as different block objects, and they are passed as

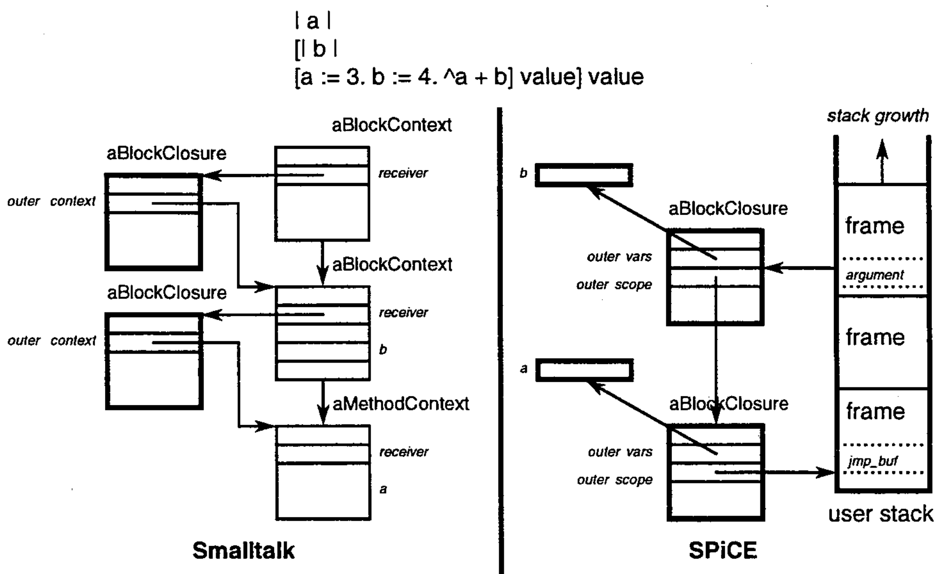


Figure 3.6: Snapshots of a block evaluation in Smalltalk and SPiCE

```
/*
The original Smalltalk code is:
(Collection)select: aBlock ifError: errorBlock
    | newCollection |
    newCollection := self species new.
    self do: [:each | | result |
                result := aBlock value: each.
                (result isKindOf: Boolean) ifFalse: [^errorBlock value].
                result ifTrue: [newCollection add: each]].
    ^newCollection
*/
#define ms_0(self, selector)        sp_lookup0(selector, P1(self))
#define ms_1(self, selector, a1)    sp_lookup1(selector, P2(self, a1))
#define sp_setContext(c_env) \
    sp_oop _resultOop; jmp_buf c_env; \
    if ((_resultOop = (sp_oop)_setjmp(c_env)) != 0) return _resultOop;

typedef struct { sp_oop blockFunction, contextArray, outerScope; } sp_BlockClosure;

sp_oop sp_m_Collection_select_ifError_(P3(self, aBlock, errorBlock))
sp_oop self, aBlock, errorBlock;
{
    /* The variable 'newCollection' can be allocated in the stack frame, because at
     * translation time the variable 'newCollection' is never assigned after a block
     * is created.
     */
    sp_oop newCollection = nil;
    sp_setContext(c_env);

    newCollection = ms_0(ms_0(self, &s_species), &s_new);
    ms_1(self, &s_do_, sp_mkBlock(
            sp_m_b1_Collection_select_ifError_,                  /* blockFunction */
            sp_mkCtxAry(3, aBlock, errorBlock, newCollection),   /* contextArray */
            c_env));                                             /* outerScope */
    return newCollection;
}

sp_oop sp_m_b1_Collection_select_ifError_(P2(_block, each))
sp_oop _block, each;
{
    /* The variables 'aBlock', 'errorBlock' and 'newCollection' can be copied into
     * the stack frame, because at translation time the variables 'aBlock',
     * 'errorBlock' and 'newCollection' are never assigned after a block is created.
     */
    sp_oop result = nil;
    sp_oop contextArray = ((sp_BlockClosure*)_block->bodyPtr)->contextArray;
    sp_oop aBlock = contextArray->bodyPtr[0];
    sp_oop errorBlock = contextArray->bodyPtr[1];
    sp_oop newCollection = contextArray->bodyPtr[2];

    result = ms_1(aBlock, &s_value_, each);
    if (ms_1(result, &sp_isKindOf_, sp_sval(sp_g_Boolean)) == true)
        ;
    else
        sp_methodReturn(_block, ms_0(errorBlock, &s_value));
        /* look up c_env from _block and call longjmp() */
    if (result == true)
        return ms_1(newCollection, &s_add_, each);
    else
        return nil;
}
```

Figure 3.7: Example of the translation of blocks

arguments of a handle:do: message to an instance of Signal. When an exception occurs, an instance of Exception is created, and it searches the prepared block for the exceptional case (or *exception block*) through the call chain, and evaluates the exception block with itself as an argument. By sending a message to the instance of Exception in the exception block, the thread of control is continued, returned, or restarted. The last two cases cause the call chain to be unwound to the handle:do: message invocation.

Our implementation of exception handling employs the user stack as an exception handling stack and the C library functions for non-local return (`setjmp` and `longjmp`) to unwind the user stack. At the invocation of the handle:do: message, a pointer to the exception block and the runtime state are saved on the user stack. The pair are chained at every invocation of the handle:do: message and the chain is kept in a process object. When an exception occurs, the exception block and the runtime state are searched through this chain, the exception block is evaluated, and if needed, `longjmp` is called with the runtime state as an argument to continue execution from the corresponding `setjmp` call.

3.7 Garbage Collection

We need a garbage collection that works without language support for garbage collection, or is suited for combining programs written in multiple languages, some of which do not offer any support for garbage collection. There is, however, *conservative garbage collection* [Boehm and Weiser 1988] that works without language support. This technique treats any data accessible to the object in the stack, registers, and the data area as potential pointers. If the data value points to a valid object, the object is assumed to be accessible. Unfortunately, this technique is not suited for pure object-oriented languages. It identifies a superset of the true pointers, so it cannot relocate an object because an integer value may happen to correspond to the

address of a valid object. Therefore, it relies on the use of a mark-and-sweep algorithm and it cannot compact the heap. In a pure object-oriented language, all data, including low-level basic data such as integers, strings, and arrays, are represented as objects. and they are dynamically allocated in the heap. Therefore, many short-lived objects are created. For example, the storage allocation rate of Smalltalk exceeds 100 Kbytes/sec, and most allocated objects will die soon. In such a situation, the mark-and-sweep algorithm causes unacceptable pauses and the non-compacting algorithm causes unacceptable fragmentation of the heap.

Generation scavenging [Ungar 1984] is generational copying garbage collection and is used in the implementation of object-oriented languages such as Smalltalk and Self. It is very well suited for object-oriented applications because its has the following advantages: high performance, non-disruptive pauses, and compaction of the heap. We try to use generation scavenging without language support. Our approach uses an object table (OT) through which all objects are referred to by object-oriented pointers (*OOPs*). If the data value points to a valid OT entry, the object the OT entry points to is assumed to be accessible. Even if an integer value happens to correspond to the address of a valid OT entry, an object can be relocated safely by changing only the content of the OT entry.

Since the OT can be used and OT entries cannot be moved, our approach has the disadvantage compared to generation scavenging: all OT entries must be scanned to recycle dead entries. To reduce the number of scans of OT entries, we divide the OT into two parts, the *new OT* and the *old OT*, and we scan only the new OT at scavenging.

Our approach is not fully conservative. Objects must be referred to through the OT. A direct pointer or an internal pointer to an object is not allowed. However, as far as using OOPs, our approach is safe.

We call our garbage collection technique *generation scavenging with ambiguous roots*. Because of its unique features, our garbage collection technique is described in more details in Chapter 4.

3.8 Application Booting

Smalltalk has no concept of system booting. The running Smalltalk system is saved as a virtual image. In SPiCE, the translated application must be booted from the generated and compiled C code. To boot the application, objects that are never created from Smalltalk code such as process queues, symbols, and a system dictionary must be created first. Such objects are either translated by the translator[5], prepared in the runtime system, or created at the time of system initialization. At the time of system initialization, all translated global variables are added into a system dictionary object and **initialize** messages are sent to all class objects that are able to respond to the **initialize** message. Fortunately, most **initialize** methods for classes could be used for booting, and only a few **initialize** methods had to be modified. Global variables, pool variables, and class variables were initialized in **initialize** methods.

3.9 Summary

Our approach to the translation is, first, to create runtime replacement classes implementing the same functionality of Smalltalk classes that are inherently part of the Smalltalk execution model, and second, to provide a garbage collection that is suitable for object-oriented applications and combining programs written in multiple languages, some of which do not offer any support for garbage collection. The creation of runtime replacement classes is based on the concept of mapping contexts onto stack frames, and mapping compiled methods onto machine code. The runtime replacement classes encapsulate the differences of the execution model between Smalltalk and C, and enables the generation of portable, efficient, and interoperable C code while preserving the functionalities of Smalltalk. Our garbage collection technique fills the gaps of the data model (storage management) between Smalltalk and C, and enables the generated C data structures (Smalltalk objects) to be mixed with other languages' data structures.

[5]The translator can translate a specified object into C data as described in Section 3.4.

We describes our garbage collection technique, called generation scavenging with ambiguous roots, in more details in Chapter 4, and we evaluate SPiCE in Chapter 5.

Chapter 4

Generation Scavenging with Ambiguous Roots

Considering the advantages of object-oriented programming such as modular design and software reuse, garbage collection is necessary to avoid introducing unnecessary inter-module dependencies. A method operating on an object should not have to depend what other methods may be operating on the same object, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when other modules are not interested in a particular object. Also, garbage collection is necessary to avoid many bugs and remove a lot of clutter from code.

In this chapter, we present a garbage collection technique that is suitable for object-oriented applications and combining programs written in multiple languages, some of which do not offer any support for garbage collection. First, Section 4.1 shows the requirements for garbage collection. Then, Section 4.2 reviews conservatism in garbage collection. In the following four sections, we present two versions of our garbage collection technique called *generation scavenging with ambiguous roots*. Section 4.7 describes implementation details of the garbage collector of SPiCE that is a practical implementation of generation scavenging with ambiguous roots. Finally, Section 4.8 summarizes our garbage collection technique.

4.1 Requirements for Garbage Collection

Recall that the goal of this work is to translate Smalltalk code into portable, efficient, and interoperable C code while preserving the functionalities of Smalltalk. Thus, our requirements for garbage collection are as follows:

1. It should work without language support for garbage collection.

   For interoperability, the generated C data structures (Smalltalk objects) must be safely mixed with other languages' data structures. Therefore, the garbage collection should work without language support, or should be suited for combining programs written in multiple languages, some of which do not offer any support for garbage collection.

2. It should compact the heap.

3. It should have a short pause time.

   In a pure object-oriented language, all data, including low-level basic data such as integers, strings, and arrays, are represented as objects, and they are dynamically allocated in the heap. Therefore, many short-lived objects are created. For example, the storage allocation rate of Smalltalk exceeds 100 Kbytes/sec, and most allocated objects will die soon. In such a situation, a non-compacting garbage collection algorithm causes unacceptable fragmentation of the heap. Also, a mark-and-sweep or stop-and-copy garbage collection algorithm causes disruptive pauses for near real-time or interactive applications.

4.2 Conservatism in Garbage Collection

4.2.1 Conservative Collector

Abstractly speaking, the basic functioning of a garbage collector consists of two part:

1. Distinguishing the live objects from the garbage in some way (*garbage detection*), and

2. Reclaiming the garbage objects' storage, so that the running program can use it (*garbage reclamation*).

In practice, these two phases may be functionally or temporally interleaved, and the reclamation technique is strongly dependent on the garbage detection technique. Most garbage collectors use a "liveness" criterion defined in terms of a *root set* and *reachability* from these roots, and they require some cooperation from the language (compiler or interpreter) for garbage detection. The garbage collector must identify pointers in roots, such as the stack and registers, or in objects.

There is, however, a technique called *conservative pointer-finding* [Boehm and Weiser 1988] that does not require any cooperation from the language. This technique treats any data accessible to the object in the stack, registers, and the data area as a potential pointer. If the data value points to a valid object, the object it points to is assumed to be accessible. This simplifies the garbage collection of programs written without garbage collection in mind, and programs written in multiple languages, some of which are uncooperative. However, this technique imposes an additional constraint on the garbage collector. It identifies a superset of the true root set, i.e., *ambiguous roots*, so it cannot relocate an object because an integer value may happen to correspond to the address of a valid object. Conservative pointer-finding is used in many conservative collectors with various garbage reclamation algorithms [Boehm and Weiser 1988; Demers *et al.* 1990; Boehm *et al.* 1991]. However, these collectors do not satisfy our requirements because they cannot compact the heap because of the constraint of conservative pointer-finding.

4.2.2 Partially Conservative Collector

Mostly-copying collector [Bartlett 1988] is a compacting "partially conservative" collector. Partially conservative means that the collector scans the stack and registers

conservatively by using conservative pointer-finding but the heap precisely, so object formats in the heap must be recognizable by the collector.

In this technique, the heap is divided into a number of equal-size pages and an object is allocated into the page. The heap is conceptually divided into two equal space: *current-space* and *next-space*, and each page has a space identifier that identifies the space that objects on the page belong to. Mostly-copying collection proceeds as follows. First, the collector scans the stack and registers conservatively by using conservative pointer-finding, that is, it guesses which pages contain objects that may be referred to from pointers in the stack and registers. These pages are *promoted* to the next-space. Promoting a page means that the space identifier of the page is flagged such that the page will be retained when the collector is over. The pointers in the stack and registers refers to such pages are ambiguous roots. It is important that the pages referred to from these ambiguous roots be locked, so the objects on these pages can be retained. Once the initial promoted objects have been identified, the collector scans the promoted objects and forwards (copies) all objects referred to from these objects into a more compact area in next-space. The forwarded pointers of the copied objects are updated in the object being scanned. Scanning continues until all promoted and forwarded objects have been scanned.

Mostly-copying collector is used in the implementation of Modula-3 [Cardelli *et al.* 1989] and SCHEME->C [Bartlett 1989]. However, it does not satisfy our requirements because it relies on the use of stop-and-copy algorithm that may cause long pauses and its compaction of the heap is marginal.

4.2.3 Reference Counting Collector

Reference counting [Collins 1960] is a garbage collection technique that uses another liveness criterion. In this technique, each object has an associated count of the references (pointers) to it. Each time a reference to the object is created, e.g., when a pointer is copied from one place to another by an assignment, the object's count is incremented. When an existing reference to an object is eliminated, the count is

decremented. The memory occupied by an object may be reclaimed when the object's count equals zero. since this indicated that no pointers to the object exist and the running program could not reach it. The problems of this technique are: (1) it has a great deal of overhead in an adjusting reference counts done by the running program: (2) it fails to reclaim circular structures: and (3) it does not compact the heap.

Deferred reference counting [Deutsch and Bobrow 1976] avoids adjusting reference counts for most short-lived pointers from the stack, and greatly reduces the overhead of reference counting. But this technique does not solve the rest of the problems.

Reference counting and deferred reference counting do not satisfy our requirements because they do not compact the heap.

4.3 Outline of Generation Scavenging with Ambiguous Roots

4.3.1 Generation Scavenging

Before describing our technique. we explains Ungar's generation scavenging [Ungar 1984]. Ungar's generation scavenging is generational copying garbage collection which satisfies our second and third requirement. This technique is very suited for pure object-oriented languages and used in the implementation of them. such as Smalltalk and Self. Fig. 4.1 shows the heap structure and a sample object structure of generation scavenging.

In generation scavenging. each object is classified as either new or old by generation. Old objects reside in a region of memory called *old space*. New objects similarly reside in *new space*. All old objects that refer new objects are registered in a table called the *remembered set*. New space is divided into three spaces: *creation space*, *past survivor space*, and *future survivor space*. All objects are created in creation space. All scavenged (copy collected) objects reside in past survivor space. No object resides in future survivor space at the running program.

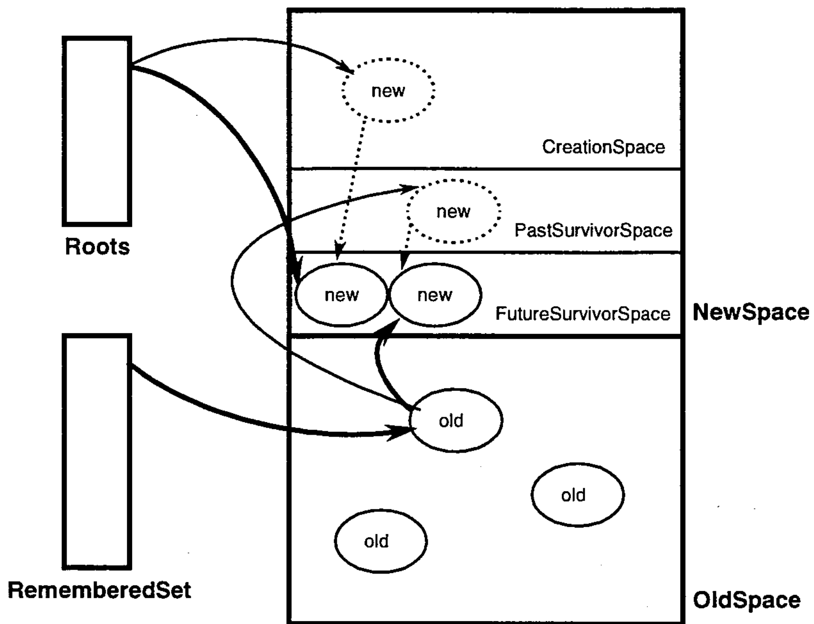


Figure 4.1: Heap structure of generation scavenging

Generation scavenging proceeds as follows. First, all new objects that are reachable from the old objects in the remembered set and the roots of the system, such as the stack and registers, are scavenged, i.e., all new live object are copied from creation space or past survivor space into future survivor space. If a new object becomes old enough, that is, it survives enough scavenges, it becomes an old object and it is promoted into old space. This promotion is called *tenuring*. At the end of scavenging, past survivor space is exchanged with future survivor space.

This technique has following advantages: (1) scavenging just the youngest generation is much faster than a full garbage collection; and (2) it can compact the heap at scavenging. However, it cannot be used with conservative pointer-finding because it is a copying collector that must know whether a value is a pointer to an object or not for moving the object and updating the pointer.

4.3.2 Conservative OOP-Finding

We try to use generation scavenging in a conservative manner, that is, the collector assumes that any value on the stack, registers, and the data area could be a potential pointer. Our approach uses an *object table* (OT) through which all objects are referred to by *object-oriented pointers* (OOPs). If the data value points to a valid OT entry, the object the OT entry points to is assumed to be accessible. Even if an integer value happens to correspond to the address of a valid OT entry, an object can be relocated safely by changing only the content of the OT entry. The basic method for checking this validity, called *conservative OOP-finding*, is as follows (see Fig. 4.2):

1. If the data value is below the lowest OT address or above the highest OT address, it is not an OOP.

2. If the offset of the data value from the lowest OT address is not a multiple of the OT entry size, it is not an OOP.

3. If the OT entry the data value points to does not refer to an object, it is not an OOP.

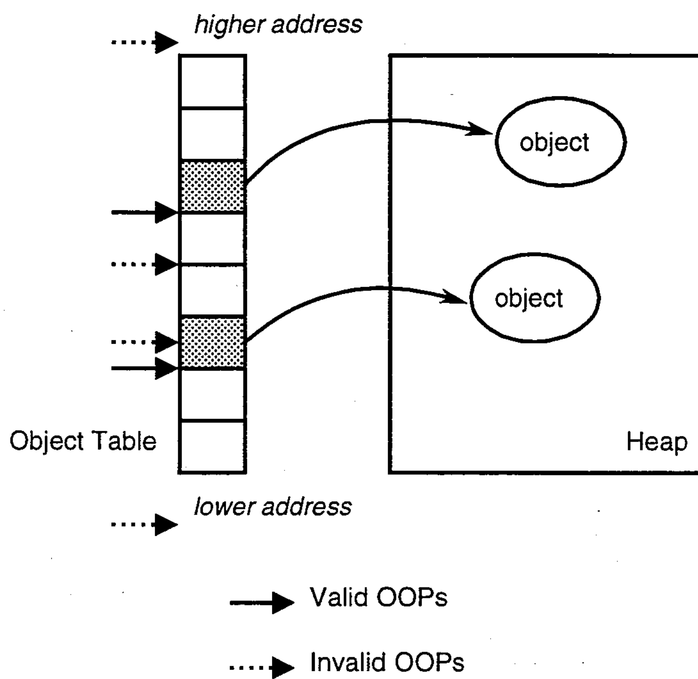


Figure 4.2: Conservative OOP-finding

Our approach is not fully conservative because objects must be referred to through the OT and a direct pointer or an internal pointer to an object is not allowed. However, as far as using OOPs, our approach does not require any cooperation from the language because the stack and registers are scanned conservatively by using conservative OOP-finding.

4.4 Collector I

We describe the collector I that is an initial version of generation scavenging with ambiguous roots. It scans not only the stack and registers but also the heap conservatively by using conservative OOP-finding, so object formats in the heap need not to be recognizable by the collector I.

Each OT entry consists of a pointer to the object's body (contents), the size of the object (size), the age of the object (age), the flag indicating that the object is new one (isNew), the flag indicating that the object is forwarded (isForwarded), and the flag indicating that the object is in the remembered set (isRemembered). The C language declaration for the OT entry is as follows:

```
typedef struct otEntry {
  word_t *contents;
  struct {
    unsigned isNew       :1;
    unsigned isForwarded :1;
    unsigned isRemembered:1;
    unsigned             :5;
    unsigned age         :8;
    unsigned size        :16;
  } gcInfo;
} *oop;
```

Fig. 4.3 shows the heap structure and a sample object structure of the collector I. The heap is divided into *CreationSpace*, *PastSurvivorSpace*, *FutureSurvivorSpace*, OT, and *OldSpace*. *OT_bottom* and *OT_top* show the lowest and the highest address

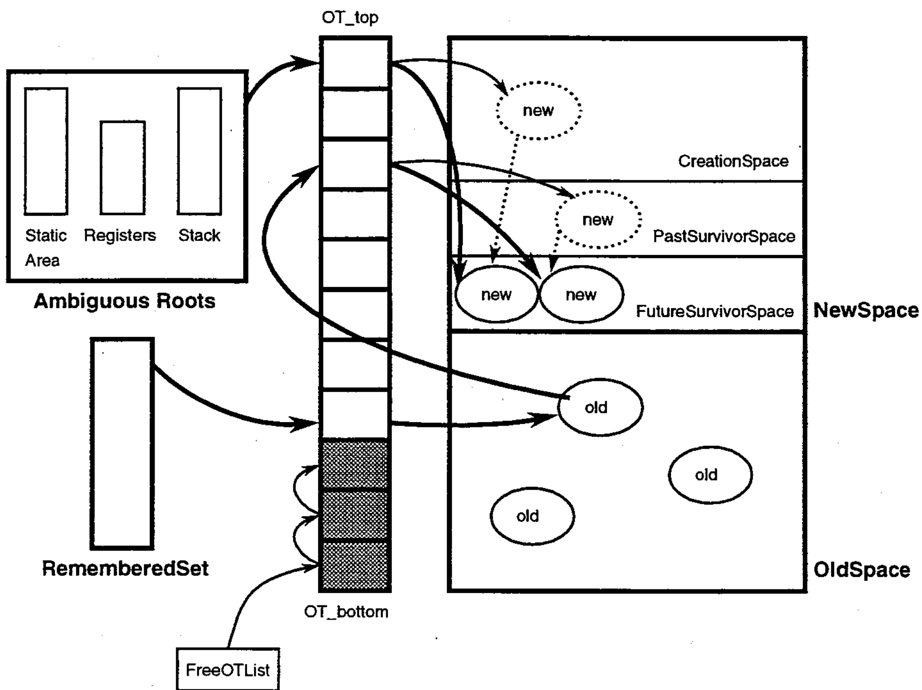


Figure 4.3: Heap structure of the collector I

of the OT. All free OT entries are managed as a linked list by using the contents field of the OT entry.

In the collector I, only new objects are scavenged, so the C language implementation of conservative OOP-finding is as follows (In this implementation, the collector I assumes that the size of the OT entry is eight bytes):

```
if ((OT_bottom <= value && value < OT_top) &&
     (value & 7) == 0 &&
     ((oop)value)->gcInfo.isNew)
{
       /* copy and forward */
}
```

Roughly speaking, the collector I reclaims objects by using Ungar's generation scavenging and reclaims OT entries by using conservative mark-and-sweep algorithm. The main differences of the collector I from Ungar's generation scavenging are to scan roots of the system by using conservative OOP-finding (ambiguous roots), to scan objects in the heap by using conservative OOP-finding, and to move an object by changing only the content of the OT entry. At the end of scavenging, all unmarked (dead) OT entries are reclaimed by sweeping all OT entries.

The complete algorithm of the collector I is shown in Appendix A.1.

4.5 Dividing Object Table

Generation scavenging is efficient because scavenging just the new generation is much faster than a full garbage collection and scavenging live objects is faster than sweeping dead ones. However, in our approach, all OT entries belonging to all generations must be swept to reclaim dead ones. This is a serious problem for efficiency because it takes a constant time at every scavenging and the pause time becomes proportional to the number of OT entries. For efficiency, the number of OT entries should be small. But the small number of OT entries causes a limitation on the number of objects used in programs.

To reduce the number of scans of OT entries, the OT is also divided into two generations, the *new OT* and the *old OT*, and the collector scans only the new OT at scavenging. The new OT has a fixed number of OT entries that usually point to new objects. The old OT has a variable number of OT entries that only point to old objects. Since new objects are never pointed to from old OT entries, new objects are scavenged by using only the new OT. When a new object is tenured into old space, the object is moved to old space and the OT entry that points to the object is also moved to the old OT. If the OT entry is referred to from the stack or registers, the OT entry is not moved because of a constraint of conservative OOP-finding. However, such a case hardly happens, because the number of objects referred to from the stack or registers are small and they are usually short-lived, i.e., not tenured.

4.6 Collector II

We describe the collector II that is a modified version of the collector I employing the divided OT. It scans the stack and registers conservatively by using conservative OOP-finding but the heap precisely to move OT entries, so object formats in the heap must be recognizable by the collector II.

The OT entry format of the collector II is nearly same as that of the collector I. In addition to the fields of the OT entry of the collector I, each OT entry has the flag indicating that the OT entry is forwarded (isOTForwarded), and the encoded object information (objInfo) which is used to interpret the object format, including the locations of pointer fields. In dynamically-typed object-oriented languages, such an information is typically a pointer to the class object that knows the format of its instances. The C language declaration for the OT entry is as follows:

```
typedef struct otEntry {
  word_t *contents;
  struct {
    unsigned isNew        :1;
    unsigned isForwarded  :1;
    unsigned isRemembered :1;
    unsigned isOTForwarded:1;
    unsigned              :4;
    unsigned age          :8;
    unsigned size         :16;
  } gcInfo;
  struct objInfo_t objInfo;
} *oop;
```

Fig. 4.4 shows the heap structure and a sample object structure of the collector II. The heap structure is the same as that of the collector I except for the OT. The OT is divided into *NewOT* and *OldOT*. *NewOT_bottom* and *NewOT_top* show the lowest and the highest address of the NewOT, and *OldOT_bottom* and *OldOT_top* show the lowest and the highest address of the OldOT. All free new OT entries and all free old OT entries are managed as a linked list respectively by using the contents field of the OT entry.

In the collector II, only new objects are scavenged, so the C language implementation of conservative OOP-finding is as follows (In this implementation, the collector II assumes that the size of the OT entry is sixteen bytes):

```
if ((NewOT_bottom <= value && value < NewOT_top) &&
    (value & 15) == 0 &&
    ((oop)value)->gcInfo.isNew)
{
      /* copy and forward */
}
```

Roughly speaking, the collector II reclaims objects by using Ungar's generation scavenging and reclaims OT entries by using partially conservative copying algorithm. The main differences of the collector II from Ungar's generation scavenging are to scan roots of the system by using conservative OOP-finding (ambiguous roots), to move an

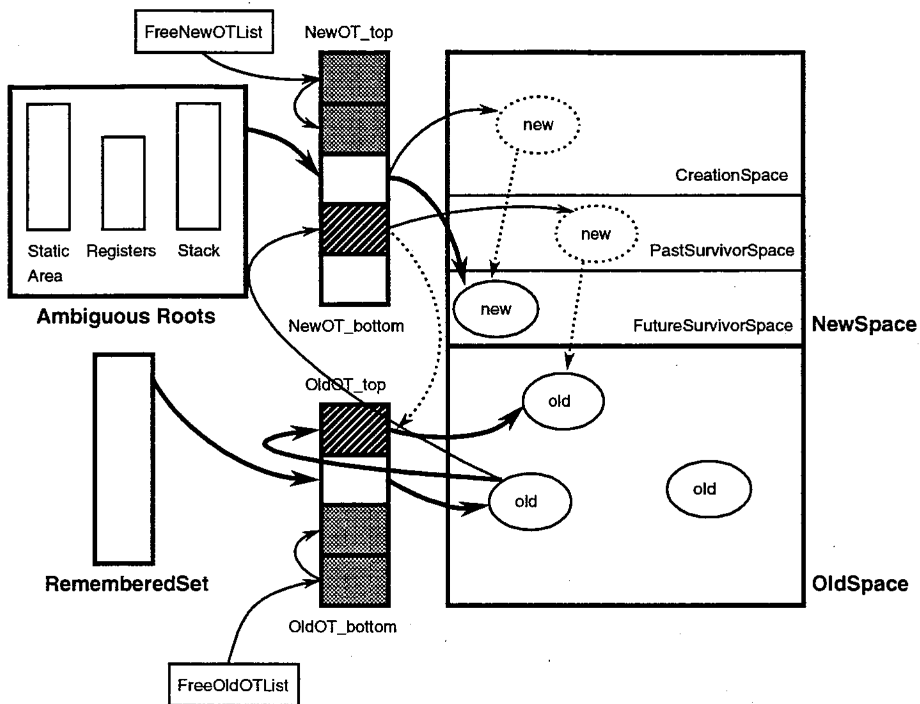


Figure 4.4: Heap structure of the collector II

object by changing only the content of the OT entry, and to tenure an object along with its OT entry if it is not referred to from ambiguous roots. Unlike the collector I, the collector II scans objects in the heap precisely which enables the new OT entries that become old enough to be moved into the old OT. At the end of scavenging, all unmarked (dead) new OT entries are reclaimed by sweeping all new OT entries.

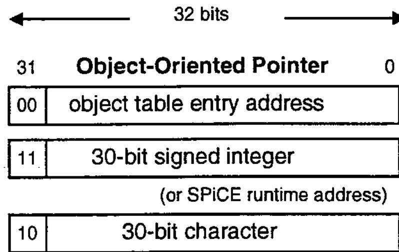
The complete algorithm of the collector II is shown in Appendix A.2.

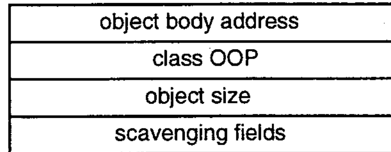
4.7 SPiCE Collector

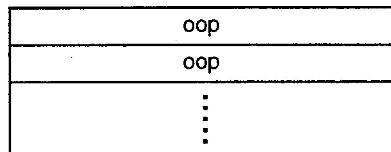
The garbage collector of the runtime system of SPiCE, called the *SPiCE collector*, is a practical implementation of generation scavenging with ambiguous roots. It adopts an algorithm of the collector II, as object formats in the heap are recognizable by the collector. We describe implementation details of the SPiCE collector.

4.7.1 Object Formats

The object format of SPiCE is very similar to that of Smalltalk. An OOP contains a high-order 2-bit tag field, used to interpret the remaining 30-bits of information: a 30-bit address of an OT entry, a 30-bit signed integer, or a 30-bit character. An OT entry is 16 bytes, four bytes each for the pointer to a body of the object, for the class OOP, for the size, and for scavenging fields that are bitfields used by the SPiCE collector. An object body is usually made up of fields containing OOPs. A body of a byte object, such as a string object, is made up of aligned 32-bit fields containing byte values. Fig. 4.5 shows the object formats of SPiCE. Pointers to SPiCE runtime functions and other objects not in the SPiCE heap, such as block functions and user stacks, are represented in a 30-bit unsigned integer using the same tag representation as an integer object. Their addresses can normally be represented in a 30-bit unsigned integer. However, for example, the text segment of a shared library in SunOS is mapped to the higher address, so the block function address is not represented in a 30-bit unsigned integer. If SPiCE runtime addresses are higher







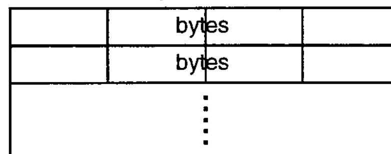


Figure 4.5: Object formats of SPiCE

than the range of a 30-bit unsigned integer, a 32-bit byte array object is used to represent the high address.

4.7.2 Intergenerational References

Generational garbage collectors must detect pointers from older to younger generations, requiring a *write barrier* (or a *store check*). That is, a program cannot simply store pointers into heap objects. Each potential store must be accompanied by checking or recording operations, to ensure that if any pointers to younger generations are created, they can be found later by the collector.

To keep track of intergenerational references from older to younger generations, we use an objectwise pointer recording scheme, i.e., the remembered set. Also the translator of SPiCE emits additional instructions along with each potential store, to perform the require write barrier operations. Because the write barrier cost is not small, optimizing the write barrier is important to overall garbage collector performance. Therefore, the translator omits the write barrier for the following cases:

1. The write barrier for storing pointers into local variables need not to be emitted because such variables are allocated on stack frames and stack frames are the roots of garbage collection.

2. The write barrier for storing immediate objects such as integers or characters need not to be emitted because storing immediate objects does not cause intergenerational references. Storing immediate objects can be identified only if the right hand of an assignment expression is a integer literal or a character literal.

3. The write barrier for storing old objects need not to be emitted because storing old objects does not cause intergenerational references. Storing old objects can be identified only if the right hand of an assignment expression is a literal, a class name, or a shared variable.

Some objects such as literal objects, class objects, and shared variable objects are allocated with their OT entries in the data area. Since old objects are never reclaimed

in generation scavenging[1], the SPiCE collector treats these objects as old objects. If these objects *refer to* new objects, they are registered in the remembered set.

4.7.3 Roots of the System

The SPiCE collector is executed using the kernel stack and it scavenges all live new objects that are reachable from the roots and the remembered set. The roots consist of the user stacks, registers, and the data area. The user stacks are scanned by using conservative OOP-finding. Registers are scanned by examining the user stacks because registers are saved on the user stacks when a process is switched or `setjmp` is called. An object is never referred to from the kernel stack because it is used only at the time of system initialization, system finalization, and garbage collection. For efficiency, the SPiCE collector has knowledge of pointer locations in the data area. Thus, it scans references only from the user stacks and the known pointer locations in the data area. Fig. 4.6 shows a memory layout in the process image of an operating system.

4.7.4 Tenuring Policy

Generational garbage collectors assume that young objects are likely to die and old ones are likely to continue to live, and concentrate the collector's efforts on the younger generations. When an object has survives some numbers of scavenges, it is moved (*tenured*) to the next older generation.

If an object lives long enough to attain tenure but then dies, that is, an object is tenured too fast, this will cause the older generation to fill up more quickly and be collected more often. Since, in generation scavenging, the old objects are no longer subject to collection, that is, they can only be collected by invoking a global mark-and-sweep garbage collector, the choice of tenuring policies is very important. We have adopted *large object area* (see Fig. 4.6) and *demographic feedback-mediated*

[1]Old objects are reclaimed using (conservative) mark-and-sweep garbage collection as in the Smalltalk system.

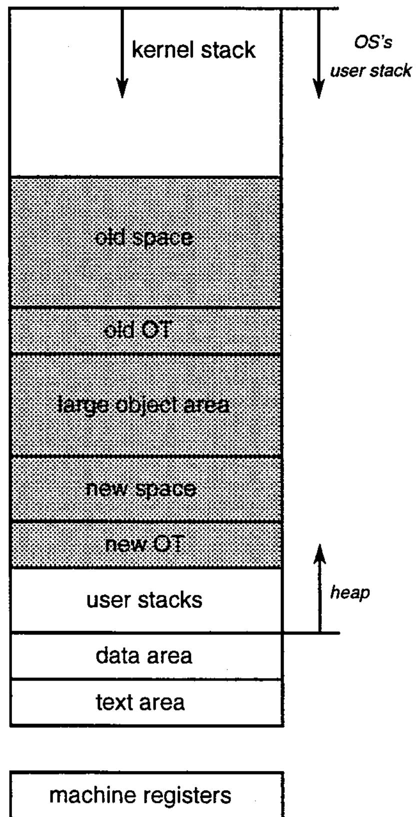


Figure 4.6: Memory layout

tenuring [Ungar and Jackson 1988; Ungar and Jackson 1992] as our tenuring policy.

A large object area keeps the data of all large objects such as bitmaps and strings. In Ungar & Jackson scheme, the headers of these large objects are stored in the youngest generation and then scavenged, but these large objects are never tenured (i.e., their headers are retained in the youngest generation until the object died), because the large object area can be made big enough to hold the data of all large objects. The SPiCE collector uses OT entries as the headers of large objects.

Ungar's generation scavenging associates with each object an explicit age which is incremented each time the object is scavenged within a generation. An object is tenured when its age count exceeds the limit (or *tenure threshold*) for the generation in which it resides. This fixed-age tenuring policy has following problems: (1) If there are very few objects being scavenged, there is no need to tenure any, but a fixed-age tenuring policy will tenure objects that are old enough anyway; (2) When the new area runs out of space in the midst of a scavenge, the remaining scavenged objects are tenured, regardless of age.

Demographic feedback-mediated tenuring policy dynamically determines the tenure threshold according to the statistics. If, after a scavenge, the survivor size are small, this policy will set the tenuring threshold to infinity for the next scavenge; no objects will be tenured the next time. If, on the other hand, the survivor size are large, then this policy will set the tenuring threshold to a value designed to tenure the excess data on the next scavenge. This value is computed by using a table indexed by age containing the number of data bytes for each age.

The Ungar & Jackson policy determines the tenuring threshold according to the size of the survived objects, but, in addition to the size, the SPiCE collector determines the tenuring threshold according to the number of the objects pointed to from the new OT. When the number of the objects pointed to from the new OT exceeds a certain threshold, the tenuring threshold is decreased to make the objects that exceed the tenuring threshold move into the old space, and freed new OT entries are made reusable.

4.8 Summary

Our technique called generation scavenging with ambiguous roots enables generation scavenging, which is very suited for pure object-oriented languages, to work without language support for garbage collection. It inherits the advantages of generation scavenging such as high performance, non-disruptive pauses, and compaction of the heap. In addition to these advantages, it enables the combination of programs written in multiple languages, some of which do not offer any support for garbage collection.

We presented two algorithms of generation scavenging with ambiguous roots: the collector I and the collector II. Both of them use conservative OOP-finding with OT. The collector I needs not to recognize object formats in the heap and reclaims OT entries by using conservative mark-and-sweep algorithm. The collector II needs to recognize object formats in the heap and reclaims OT entries by using partially conservative copying algorithm. The collector II is less conservative than the the collector I, but the collector II is more efficient.

The SPiCE collector is a practical implementation of the collector II and enables the generated C data structures (Smalltalk objects) to be mixed with other languages' data structures. We evaluate the effectiveness of the SPiCE collector in Section 5.4.

Chapter 5

Evaluation of SPiCE

Recall that issues on the translation are as follows:

1. It may impose some restrictions on the functionality of pure object-oriented languages since a clean translation from a pure object-oriented language to a conventional language is difficult due to a language gap.

2. For preserving the functionality of pure object-oriented languages, the generated code may be so stylized that it is neither portable nor interoperable with code written in other languages.

3. The two-stage translation into machine code may result in inefficient code.

In this chapter, we evaluate SPiCE with respect to these issues. First, Section 5.1 discusses the restrictions on Smalltalk that SPiCE imposes. Then, Section 5.2 discusses interoperability of the generated C code. Following this, Section 5.3 shows the performance of the generated C code compared to the fastest Smalltalk implementation, and Section 5.4 shows the performance of the garbage collector of SPiCE. Finally, Section 5.5 compares our work with related work.

5.1 Restrictions on Smalltalk

Smalltalk has some reflective facilities; programs can examine their activation records and can redefine their method [Foote and Johnson 1989]. However, these reflective facilities are not usually used in the application programming and they are only used in the exploratory programming. For example, these facilities are used to construct a backtracking facility in Smalltalk [Lalonde and Gulik 1988]. Preserving the reflective facilities are beyond the translation approach and all translators including SPiCE impose a restriction on the reflective facilities. Another restriction of SPiCE is that a modification of the original classes that correspond to the runtime replacement classes is not permitted. Since these classes are inherently part of the Smalltalk execution model, the modification of them easily results in a crash of the Smalltalk system, so most applications do not modify them.

Our experiences with SPiCE show that these restrictions do not interfere with the application programming and SPiCE can translate existing Smalltalk applications as they are. Currently, five large and practical Smalltalk applications have been translated by SPiCE. All of them were translated into C without modifying the application code. These applications are:

- Smalltools that is a Mail and News browser having 97 application classes;
- SmallWalker[1] that is a World-Wide Web browser having 164 application classes (See Fig. 5.1);
- a production line simulator having 31 application classes;
- a network traffic monitoring tool having 112 application classes; and
- ObjectCast that is a commercial CASE tool supporting an object-oriented software development having 195 application classes.

[1]SmallWalker is written by the author of this thesis and it is available from `ftp://st.cs.uiuc.edu/pub/Smalltalk/st80_r41/SmallWalker1.0/`.

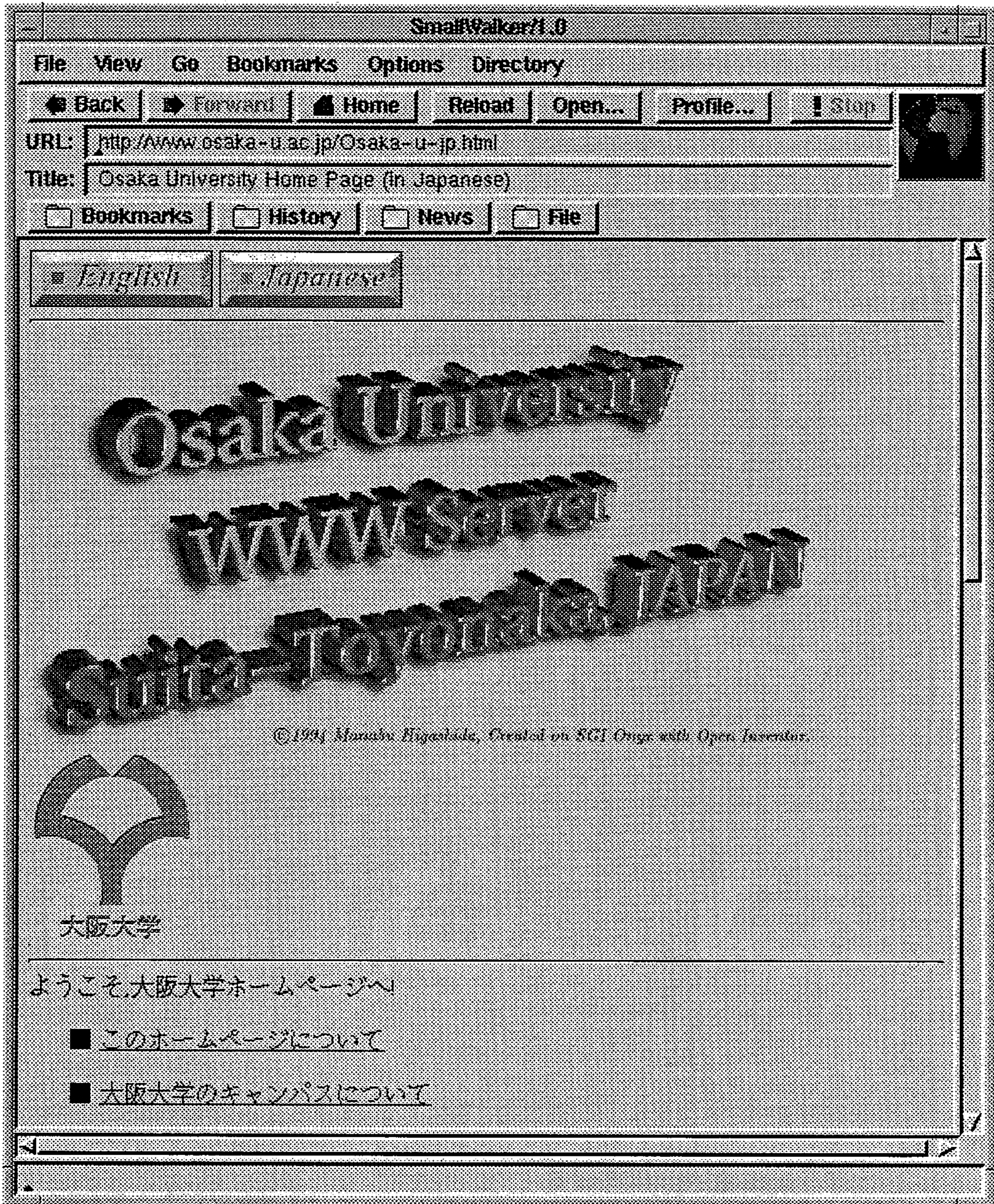


Figure 5.1: SmallWalker: A Smalltalk WWW browser

The translator of SPiCE is implemented on ParcPlace Smalltalk Release 4.1. Its runtime system is implemented in 39,000 lines of C code and 1600 lines of assembly code, and runs on the SunOS (SPARC-based machine) and the Linux (Intel 386-, 486-, Pentium-based machine).

Portability of the generated C code was tested between different processors and between different C compilers. The order of the arguments evaluated on a SPARC-based machine is contrary to a Intel 386-based machine. The generated C code was compiled and ran correctly under the SunOS using both Sun C compiler (non ANSI C compiler) and GNU C compiler (ANSI C compiler). The same code was also compiled and ran correctly under the Linux using GNU C compiler.

The translator and the runtime system are very stable. Smalltools and Small-Walker translated by SPiCE are used daily by many researchers at Fuji Xerox Co., Ltd. under the SunOS and the Linux.

5.2 Interoperability with C

SPiCE generates an object-oriented stylized C code, that is, all objects are referred to from OOPs and all computation are performed by sending messages to objects. However, the generated C code uses as much of C facilities as possible: the generated C code follows the C procedure call/return mechanism and the generated C data structures (Smalltalk objects) can be safely referred to from C variables. Therefore, the generated C code is easy to interoperate with other C programs or code written in other languages.

We show a small example of the combination of code originally written in Smalltalk with code written in C. Fig. 5.2 shows a sorting C program using the translated Smalltalk class libraries. First, a SortedCollection object is created (line 8). Then, strings are read from the terminal, converted into String objects, and added to the SortedCollection object (line 9–12). Finally, the contents of the SortedCollection object are copied into character array and printed in order (line 13–18). Sorting is done by

```
 1 sort()
 2 {
 3   extern sp_asc sp_g_SortedCollection;
 4   extern sp_symbol s_new, s_add_, s_at_;
 5   sp_oop collection, string;
 6   int i, size;
 7   char line[1024];

 8   collection = ms_0(sp_g_SortedCollection.asc_value, &s_new);

 9   while(gets(line) != NULL) {
10     ms_1(collection, &s_add_, sp_STString(line));
    /* or,
11   * sp_m_SortedCollection_add_(P2(collection, sp_STString(line)));
     */
12   }

13   size = sp_oopIntVal(ms_0(collection, &s_size));
14   for (i = 1; i <= size; i++) {
15     string = ms_1(collection, &s_at_, sp_asSmallInt(i));
16     sp_copyToCString(string, line, sizeof(line));
17     printf("%s\n", line);
18   }
19 }
```

Figure 5.2: Combination of Smalltalk and C

the SortedCollection object. This SortedCollection object is not reclaimed by garbage collection during the execution of this function because it is referred to from a stack frame. If a programmer knows the type (class) of a message receiver, the programmer can directly call a function corresponding to the method invoked by the message. For example, line 11 can be substituted for line 10.

We also show a more practical example, a Xebec document class browser (See Fig. 5.3). Xebec is a document database management system that can handle multiple document types and multiple document architectures [Nakatsuyama *et al.* 1995; Kyojima and Yasumatsu 1995]. Xebec also manages document schemata of the Xebec data model and document classes that are logical structures expressed in document architectures, such as *document type definition* (DTD) of *standard generalized markup language* (SGML) [International Standardization Organization 1986]. Xebec is implemented on the ObjectStore database system [Lamb *et al.* 1992] and written in OC++ that is a persistent C++ for the ObjectStore. Xebec document class browser can search and browse persistent document classes with a graphical user interface. The combination of code originally written in Smalltalk with code written in OC++ is achieved by an external C interface of OC++, and the design of the combination is based on *proxy pattern*, which provides a surrogate or placeholder for another object to control access to it [Gamma *et al.* 1994]. ObjectStore is very critical to memory access because ObjectStore loads persistent objects into virtual memory by using page fault. However, our garbage collection can coexist with the ObjectStore mechanism.

5.3 Performance in C

We compared the performance of the generated C code with the ParcPlace Smalltalk implementation. The ParcPlace Smalltalk implementation called *HPS* is currently the fastest Smalltalk implementation and uses the *Deutsch-Schiffman technique* [Deutsch and Schiffman 1984]. This technique dynamically translates compiled methods into

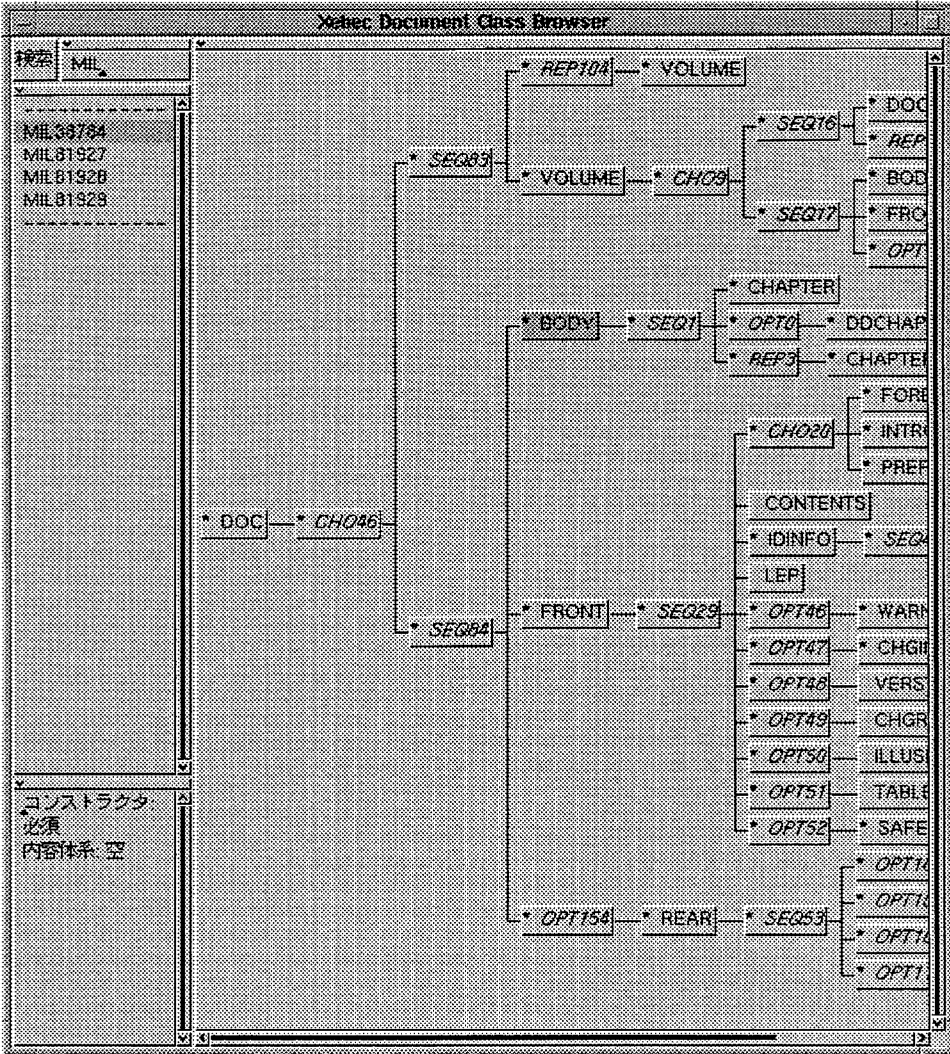


Figure 5.3: Xebec document class browser

machine code and maps contexts onto frames on stacks prepared in the virtual machine. The runtime system of SPiCE uses primitives of HPS except for processes, blocks, and exception handling.

We used the following four programs to compare the performance: *Smalltalk system benchmarks* [McCall 1983], *Richards operating system simulation benchmark* [Chambers *et al.* 1989], Lisp interpreter, and color image quantization using ordered dither. We first measured the performances of all programs on HPS, and translated all programs using SPiCE, then measured the performance of the generated C code. We used a Sun4/110 (SPARC 14.28MHz) with 20 Mbytes memory, and a Sun C compiler with optimization level 2 (with `-O2` option).

Table 5.1 presents the results of the Smalltalk system benchmarks. The first two columns show the performance rates of SPiCE and HPS compared with *Dorado* [Deutsch 1983]. The third column shows the ratio of SPiCE to HPS, and the larger numbers indicate the better performance of SPiCE. The top four benchmarks in table 5.1 are called *macro-benchmarks*, and these test the performance of the Smalltalk system at high-level activities. The rest of the benchmarks are called *micro-benchmarks*, and these test the basic bytecodes and primitives.

In macro-benchmarks that show the performance at the application level, SPiCE runs between 0.95 times to 1.48 times the speed of HPS. This shows that the translated application by SPiCE runs at roughly the same speed as HPS. In micro-benchmarks, the results of SPiCE and HPS are very different due to their implementations. Some of the lower performances of SPiCE in micro-benchmarks result from the in-line primitives of HPS. HPS translates arithmetic and relational operations in-line with a call to a runtime routine if the operands are not SmallInteger. The loading of temporary variables in SPiCE is much faster than in HPS because temporary variables are translated into those in C.

Table 5.2 presents the execution times of the other benchmark programs. The third column shows the ratio of HPS to SPiCE, and the larger numbers indicate the better performance of SPiCE. SPiCE runs between 0.93 times to 1.78 times the

Table 5.1: Smalltalk system benchmarks

Benchmark	*SPiCE* (rate)	*HPS* (rate)	*SPiCE/HPS* (ratio)
KeyboardLookAhead	326.923	274.194	1.19
KeyboardSingle	330.804	283.117	1.17
TextDisplay	336.842	355.556	0.95
TextEditing	329.687	223.28	1.48
AsFloat	836.667	836.667	1.0
BasicAt	184.0	153.333	1.2
BasicAtPut	179.832	164.615	1.09
FloatingPointAddition	492.0	1230.0	0.4
Perform	161.111	120.833	1.33
StringReplace	10260.0	10260.0	1.0
TextScanning	208.0	208.0	1.0
LoadInstVar	153.631	74.3243	2.07
LoadLiteralIndirect	1593.33	199.167	8.0
LoadLiteralNRef	2.75e7	458.333	60000.0
LoadQuickConstant	4.91e7	446.364	110000.0
LoadTempNRef	2.75e7	275.0	100000.0
LoadTempRef	3.9e7	327.731	119000.0
PopStoreInstVar	96.648	24.7143	3.91
PopStoreTemp	92.5532	133.846	0.69
3div4	1122.0	561.0	2.0
3lessThan4	89.5522	163.636	0.55
3plus4	100.0	265.0	0.38
3times4	76.2548	119.697	0.64
ActivationReturn	711.429	553.333	1.29
ShortBranch	1.23e7	1.23e7	1.0
WhileLoop	69.5161	46.3441	1.5
ArrayAt	187.0	187.0	1.0
ArrayAtPut	156.429	115.263	1.36
Size	240.0	180.0	1.33
StringAt	261.429	305.0	0.86
StringAtPut	253.333	285.0	0.89
Class	630.0	315.0	2.0
Creation	89.2031	150.87	0.59
EQ	598.361	214.706	2.79
PointCreation	193.651	457.5	0.42
PointX	164.762	150.435	1.1
StreamNext	495.556	371.667	1.33
StreamNextPut	490.521	398.077	1.23
Value	177.857	166.0	1.07

Table 5.2: Benchmark running times

Benchmark	*SPiCE* (ms)	*HPS* (ms)	*HPS/SPiCE* (ratio)
Richards Benchmark	8225	9232	1.12
Lisp Interpreter	27366	48802	1.78
Quantization	8485	7900	0.93
Modified Quantization§	7310	—	1.08

§The translated (Depth1Image) rowAt:putAll:startingAt: C function is modified

speed of HPS. These results are similar to those of the macro-benchmarks. The performance of SPiCE using the Lisp interpreter is 1.78 times faster than HPS. In the Lisp interpreter, a stack grows very deep because of the recursive evaluation of S-expressions. In HPS, when the stack becomes full, frames on the stack are flushed out and converted into context objects, degrading the performance. In color image quantization, HPS is faster because the percentage of arithmetic operations in this program is large. We modified the translated C function **rowAt:putAll:startingAt:** in the **Depth1Image** class, which is most time-consuming, to make use of C arithmetic operations instead of runtime arithmetic calls, and this made SPiCE 1.08 times faster than HPS.

Table 5.3 presents an execution profile of the runtime system using Smalltools, described in Section 5.1, for three hours. The method lookup took 20.3 percent of the total time. The runtime system checks external events such as keyboard input at the time of method lookup. Arithmetic operations took 10.8 percent and scavenging took 4.9 percent of the total time. Scavenging was executed 8051 times, and this shows that the time for one scavenging execution is 551.21 (sec) / 8051 = 68 (msec), and the interval between scavengings is 11285.4 (sec) / 8051 = 1.4 (sec). The pause times for garbage collection are limited to about 68 msec, which is short enough to run near real-time or interactive applications.

Table 5.3: SPiCE runtime profile

Name	*running time* (sec)	*percentage* (%)
Total	11285.4	—
MethodLookup	2288.56	20.3
Arithmetic	1221.23	10.8
Scavenging	551.21	4.9
Block Creation	438.61	3.9
Accessing	302.86	2.7

number of times scavenging was executed:	8051 times
time for one scavenging execution:	68 msec
interval between scavengings:	1.4 sec

5.4 Performance of Garbage Collection

We measured the performance of the SPiCE collector on a Sun SPARCStation 1+ (SPARC 25MHz) with 20 Mbytes memory. The size of an OT entry is 16 bytes and the size of the new OT is 320 Kbytes (20 K entries). The new space consists of a creation space and two survivor spaces. The size of the creation space is 700 Kbytes, and the size of the two survivor spaces is 150 Kbytes each, making the total size of the new space 1000 Kbytes. The size of the old space including the old OT is 1000 Kbytes, and the size of the large object area is 1000 Kbytes. The total size of the heap is 3320 Kbytes.

We used the following three programs to measure the performance of the SPiCE collector: Lisp interpreter (lisp), Smalltalk parser (parser), and the Smalltalk environment (environment). All programs are originally written in Smalltalk and then translated into C by SPiCE. With the Lisp interpreter, we ran a tower of hanoi. With the Smalltalk parser, we parsed a whole source code of the Smalltalk system, the size of it is 2222520 bytes. With the Smalltalk environment, we edited text files and played games during an interactive run for an hour.

Table 5.4 shows the results of the measurements of the performance of the SPiCE

Table 5.4: Performance of the SPiCE collector

program	lisp	parser	environment
total execution time (sec)	88.0	109.4	3779.6
garbage collection time (%)	3.9	1.5	6.9
pause time (msec)	54	46	97
for scavenging	12	4	55
for OT reclamation	42	42	42
number of created objects	138576	689383	52681486
size of created objects (Kbytes)	25976.5	17154.4	619074.0
storage allocation rate (Kbytes/sec)	295.2	156.8	163.8
number of new objects referred to from the stacks	156	28	43
number of old references from new OT	19	19	23

collector. The ratio of garbage collection time to the total execution time was between 1.5 % to 6.9 %. The pause time of garbage collection was between 46 msec to 97 msec, and this is short enough to run near real-time or interactive applications. The OT reclamation takes a constant time (42 msec) in all cases and it spends between 43 % (42/97) to 91 % (42/46) in the pause time. This result shows the usefulness of the division of the OT into two generations.

The storage allocation rate was between 156.8 Kbytes/sec to 295.2 Kbytes/sec. At such a high storage allocation rate, the total heap size, which was 3320 Kbytes, was enough to run these programs. This result shows the effectiveness of the compaction of the heap.

The average number of new objects referred to from the stacks at scavenging was between 28 to 156, and the ratio of it to the total number of new OT entries was under 0.8 (156/20K) % in all cases. The number of old references from the new OT at the end of the execution of the program was between 19 to 23, and the ratio of it to the total number of new OT entries was under 0.2 % (23/20K) in all cases. These results show that when new objects are tenured into the old space, almost all new OT entries that point to them can be moved to the old OT together.

Table 5.5 shows the results of the measurements of the runtime overhead of the garbage collection. There are two runtime overheads: indirect referencing through the

Table 5.5: Runtime overhead of garbage collection

program	lisp	parser	environment
number of indirect referencing	11021343	47408450	571588028
average cycle number of LD instruction	2.72	2.36	(3.0)
indirect referencing time (%)	1.4	4.1	(1.8)
number of store checking	131447	2288164	2022754
store checking time (%)	0.1	1.1	0.0
total overhead time (%)	1.5	5.2	(1.8)
total overhead time including gc (%)	5.4	6.7	(8.7)

OT and the write barrier to keep track of references from older to younger generations.

The runtime overhead of indirect referencing was calculated from a number of indirect referencing and an average cycle number of LD instruction. The runtime overhead of one indirect referencing (one memory indirection) corresponds to one LD instruction of SPARC. In SPARCStation 1+, if memory cache is hit, LD instruction takes 2 cycles, otherwise, it takes 15 cycles. So, we measured a hit rate of memory cache by using *Spa* [Irlam 1991] and calculated the average cycle number of LD instruction. With the Smalltalk environment, we were unable to use Spa because a program runs at 600 times slower by using Spa. Since the average cycle number of LD instruction in the Lisp interpreter was 2.72 and that in the Smalltalk parser was 2.36, we assumed that the average cycle number of LD instruction in the Smalltalk environment was 3.

The ratio of the total overhead time to the total execution time was between 1.5 % to 5.2 %, and even if the garbage collection time is added, it was between 5.4 % to 8.7 %. This results shows the effectiveness of the SPiCE collector even though it has some runtime overheads for garbage collection. Incidentally, in an experimental evaluation, conservative mark-and-sweep collection spends between 10.6 % to 19.0 % of the total execution time, and conservative generational collection spends between 8.9 % to 19.2 % of the total execution time [Boehm *et al.* 1991].

5.5 Comparison with Related Work

Through the evaluation, SPiCE has proven to be able to generate portable, efficient, and interoperable C code, and to impose minimal restrictions on Smalltalk which enables the translation of existing Smalltalk applications as they are. None of the other Smalltalk translators has succeeded to do such translation. Producer, Smalltalk application compilers, and Orchard impose much restrictions on Smalltalk. Babel and Smalltalk/X are not suited for interoperating with programs written in other languages. All of them generate inefficient code less than the fastest Smalltalk implementation.

The C interoperability of Smalltalk has recently been improved. *C Programming Objectkit* [Par 1992] from ParcPlace enables Smalltalk to inter-call procedures with C. This C language interface has two disadvantages compared with our approach. First, inter-calling procedures with C has the same performance overhead as the primitive calls in the Smalltalk system. To call-out to the C procedure, the runtime state of Smalltalk must be saved, then the runtime state of C must be restored, and vice versa. Second, programmers must be careful of memory management. A C pointer to a Smalltalk object cannot be maintained across call-backs. Our approach can maintain a C pointer to a translated Smalltalk object owing to conservative OOP-finding.

The Deutsch-Schiffman technique is used to implement the virtual machine, but it is interesting to compare it with our work. This technique dynamically maps compiled methods and contexts onto machine code and stack frames while our system maps them statically. The performance advantages of dynamic translation over static translation are that it can translate arithmetic and relational operations in-line, which can be written at assembler level, and that it can use in-line cache that modifies the machine code at runtime. SPiCE executes applications at roughly the same speed as ParcPlace Smalltalk using dynamic translation. It seems to us that this result is not too bad. None of the other Smalltalk translators could run the translated application at the same speed as the ParcPlace Smalltalk.

Chapter 6

Conclusion

6.1 Summary of Main Results

This thesis has proposed a translation method from Smalltalk into interoperable C code. The usefulness and effectiveness of the proposed method have been evaluated and the proposed method has proven to be able to generate portable, efficient, and interoperable C code, and to impose minimal restrictions on Smalltalk which enables the translation of existing Smalltalk applications as they are. For example, five large and practical Smalltalk applications, including one commercial application, have been translated without modifying the application code. Moreover, a practical application written in Smalltalk and a persistent C++ has been developed. The performance of the generated C code is roughly the same as the fastest Smalltalk implementation.

In this thesis, we have proposed the design and implementation of SPiCE that is a system for translating Smalltalk into C. The key feature of the translation is a creation of runtime replacement classes, which implements the same functionality of Smalltalk classes that are inherently part of the Smalltalk execution model. The creation of runtime replacement classes is based on the concept of mapping activation record objects of Smalltalk onto stack frames, and mapping compiled code of Smalltalk onto machine code. The runtime replacement classes encapsulate the differences of the execution model between Smalltalk and C, and they enable the generation of

portable, efficient, and interoperable C code while preserving the functionalities of Smalltalk.

We have also proposed a new garbage collection technique called generation scavenging with ambiguous roots, which is suitable for object-oriented applications and combining programs written in multiple languages, some of which do not offer any support for garbage collection. The key idea of our garbage collection technique is the use of conservative stack scanning and indirect referencing together. This technique fills the gaps of the data model (storage management) between Smalltalk and C, and enables the generated C data structures (Smalltalk objects) to be mixed with other languages' data structures. Our garbage collection spends less than 7 % of the total execution time, and even if other runtime overheads for garbage collection are included, it spends less than 9 % of the total execution time.

Our technique of the runtime replacement classes is rather specific to the Smalltalk language and environment, but the concept of mapping activation record objects onto stack frames and mapping compiled code onto machine code is helpful to implement a translator for other high level languages such as Self and Scheme.

Our garbage collection technique is applicable to other languages' implementation or runtime system. The key idea of our garbage collection, that is, the use of conservative stack scanning and indirect referencing together, allows the implementation of other kinds of copying or compacting garbage collection in a conservative manner.

SPiCE has proven to be practical and stable through the daily use of translated applications by many researchers at Fuji Xerox Co., Ltd..

6.2 Future Work

An issue of SPiCE is that it executes translated applications roughly at the same speed as those of the original Smalltalk. At the start of the SPiCE project, we thought that a straightforward translation of Smalltalk into C would result in a performance improvement, but through further studies it turns out that the straightforward translation

does not improve the performance and other optimization techniques are required. However, SPiCE has enough room to use other optimization techniques.

There are some optimization techniques that can be used in SPiCE:

- *Compressed dispatch tables*

  Compressed dispatch tables implement message dispatching with a runtime overhead of one memory indirection plus an equality test. The technique is similar to virtual function table lookup of C++, but it reduces the space requirements of dispatch tables. It can offer constant time performance. Many algorithms for compressing dispatch tables have been proposed [Dixon *et al.* 1989; Andre and Royer 1992; Driesen 1993; Amiel *et al.* 1994; Driesen and Holzle 1995]. These optimization techniques can be used in SPiCE.

- *Static type prediction*

  The Self compiler uses *customized compilation* that generates multiple versions of machine code according to the type of receiver of a message. Because the type of receiver can be identified only at execution time, customized compilation cannot be used in SPiCE as is. However, when the type of receiver is unknown, the Self compiler uses *static type prediction* to generate better code for some common situations. Certain messages are more likely to be sent to some types of receivers than others: arithmetic messages such as +, −, and < are likely to be sent to integers. Thus, the Self compiler statically predicts types and generates optimized code including runtime type tests to verify its prediction. This optimization technique can be used in SPiCE.

- *Type inferencing*

  Identifying the class of the receiver, i.e. type inferencing, is very difficult in Smalltalk [Suzuki 1981: Borning and Ingalls 1982]. However, recently the *cartesian product algorithm* [Agesen 1995] has been proposed. It can infer concrete types of object-oriented programs, and by incorporating it into the Self compiler, it can in-line over 95 % of all message sends. Since the Self language and

environment is very similar to the Smalltalk language and environment, this algorithm can be adopted to SPiCE.

Another issue of SPiCE is that it does not support identification of the classes and methods required to execute an application. Therefore, the classes to be translated must be specified by the user and all the methods in the translated classes are translated. To reduce the size of translated applications, the classes and methods required must be identified. The size of the text area occupied more than 70 percent of the file size of the translated application, so identifying the methods is very important. One approach to reduce the size is to use the cartesian product algorithm. This algorithm was also used to extract small and self-contained applications from the Self programming environment. Another approach is to add a modular facility to Smalltalk like Modular Smalltalk [Wirfs-Brock and Wilkerson 1988], which makes an application into a collection of modules and clarifies the classes and methods required to execute the application.

For interoperability, our approach addresses the shared data structures, the shared address space, and the shared threads of control. The problem of shared data representation is beyond the scope of our approach. On the other hand, the RPC-based approaches have made much efforts on the shared data representation. For example, recent RPC-based approaches such as Horus [Gibbons 1987] and Matchmaker [Jones *et al.* 1985] automate the generation of code that maps the representations of data types between one language/machine and another via an *interface description language* (IDL). Their results of the efforts such as IDL should be incorporated in our approach.

Bibliography

[Agesen 1995] O. Agesen: *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, December 1995.

[Amiel *et al.* 1994] E. Amiel, O. Gruber, and E. Simon: "Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables," in *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA) '94*, pp. 244–258, October 1994.

[Andre and Royer 1992] P. Andre and J.-C. Royer: "Optimizing Method Search with Lookup Caches and Incremental Coloring," in *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pp. 110–126, October 1992.

[Bartlett 1988] J. F. Bartlett: "Compacting Garbage Collection with Ambiguous Roots," WRL Research Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, February 1988.

[Bartlett 1989] J. F. Bartlett: "SCHEME->C: a Portable Scheme-to-C Compiler," WRL Research Report 89/1, Digital Equipment Corporation Western Research Laboratory, Palo Alto, February 1989.

[Bershad *et al.* 1987] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislao, and M. Schwartz: "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," *IEEE Transactions on Software Engineering*, Vol. 13, No. 8, pp. 880–894, August 1987.

[Birrell and Nelson 1984] A. D. Birrell and B. J. Nelson: "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39–59, February 1984.

[Bobrow *et al.* 1988] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon: "Common Lisp Object System Specification X3J13," *ACM SIGPLAN Notices*, Vol. 23, September 1988. (Special Issue).

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

[Boehm *et al.* 1991] H. J. Boehm, A. J. Demers, and S. Shenker: "Mostly Parallel Garbage Collection," in *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pp. 157–164, June 1991.

[Borning and Ingalls 1982] A. H. Borning and D. H. H. Ingalls: "A Type Declaration and Inference System for Smalltalk," in *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 133–139, January 1982.

[Cardelli *et al.* 1989] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson: "Modula-3 Report (revised)," tech. rep., Digital Equipment Corporation System Research Center, October 1989.

[Chambers *et al.* 1989] C. Chambers, D. Ungar, and E. Lee: "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes," in OOPSLA [OOPSLA 1989], pp. 49–70.

[Cla 1994] Claus Gittinger Development & Consulting: *Smalltalk/X Documentation*, 1994. Available from http://www.informatik.uni-stuttgart.de/stx/doc/online/english/TOP.html.

[Collins 1960] G. E. Collins: "A Method for Overlapping and Erasure of Lists," *Communications of the ACM*, Vol. 2, No. 12, pp. 655–657, December 1960.

[Cox and Schmucker 1987] B. J. Cox and K. J. Schmucker: "Producer: A Tool for Translating Smalltalk to Objective-C," in OOPSLA [OOPSLA 1987], pp. 423–429.

[Cox 1986] B. J. Cox: *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[Demers *et al.* 1990] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker: "Combining Generational and Conservative Garbage Collection: Framework and Implementations," in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 261–269, January 1990.

[Deutsch and Bobrow 1976] L. P. Deutsch and D. G. Bobrow: "An Efficient, Incremental, Automatic Garbage Collector," *Communications of the ACM*, Vol. 19, No. 9, pp. 522–526, September 1976.

[Deutsch and Schiffman 1984] L. P. Deutsch and A. M. Schiffman: "Efficient Implementation of the Smalltalk System," in *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 297–302, January 1984.

[Deutsch 1983] L. P. Deutsch: *The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture*, pp. 113–126. Addison-Wesley, 1983.

[Dixon *et al.* 1989] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan: "A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance," in OOPSLA [OOPSLA 1989], pp. 211–214.

[Driesen and Holzle 1995] K. Driesen and U. Holzle: "Minimizing Row Displacement Dispatch Table," in *Conference on Object Oriented Programming Systems. Languages and Applications (OOPSLA) '95*, pp. 141–155, October 1995.

[Driesen 1993] K. Driesen: "Selector Table Indexing and Sparse Arrays," in *Conference on Object Oriented Programming Systems. Languages and Applications (OOPSLA) '93*, pp. 259–270, September 1993.

[Foote and Johnson 1989] B. Foote and R. E. Johnson: "Reflective Facilities in Smalltalk-80," in OOPSLA [OOPSLA 1989], pp. 327–335.

[Gamma *et al.* 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Gibbons 1987] P. B. Gibbons: "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," *IEEE Transactions on Software Engineering*, Vol. 13, No. 1, pp. 77–87, January 1987.

[Goldberg and Robson 1983] A. Goldberg and D. Robson: *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[Goldberg 1984] A. Goldberg: *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[International Standardization Organization 1986] : "Information Processing Systems – Text and Office Systems – Standard Generalized Markup Language (SGML)." ISO 8879. International Standardization Organization, 1986.

[Irlam 1991] G. Irlam: "Spa - SPARC performance analysis package," 1991.

[Jones *et al.* 1985] M. B. Jones, R. F. Rashid, , and M. R. Thompson: "Matchmaker: An Interface Specification Language for Distributed Processing," in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985.

[Keene 1989] S. E. Keene: *Object-Oriented Programming in Common Lisp – A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[Kernighan and Ritchie 1978] B. W. Kernighan and D. M. Ritchie: *The C Programming Language.* Prentice-Hall, 1978.

[Kyojima and Yasumatsu 1995] M. Kyojima and K. Yasumatsu: "The Logical Structure Translation for Xebec Document Database Management System," in *IEICE Technical Report*, Vol. 94 of *Office System*, pp. 13–18, March 1995. OFS 94-53 (in Japanese).

[Lalonde and Gulik 1988] W. R. Lalonde and M. V. Gulik: "Building a Backtracking Facility in Smalltalk Without Kernel Support," in OOPSLA [OOPSLA 1988], pp. 105–122.

[Lamb *et al.* 1992] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb: "The Object-Stote Database System," *Communications of the ACM*, Vol. 34, No. 10, pp. 50–63, October 1992.

[McCall 1983] K. McCall: *The Smalltalk-80 Benchmarks*, pp. 153–174. Addison-Wesley, 1983.

[Moore *et al.* 1994] I. Moore, M. Wolczko, and T. Hopkins: "Babel - A Translator from Smalltalk into CLOS," in *Proceedings of TOOLS USA '94*, 1994.

[Nakatsuyama *et al.* 1995] H. Nakatsuyama, M. Kyojima, Y. Okumura, K. Yasumatsu, T. Ando, G. Uchida, K. Chiba, K. Numata, and N. Kamibayashi: "An Overview of Xebec Document Database Management System," in *IPSJ SIG Notes*, Vol. 95 of *Database System*, pp. 97–104, January 1995. 95-DBS-101 (in Japanese).

[Nash and Haebich 1991] C. Nash and W. Haebich: "An 'Accidental' Translator from Smalltalk to ANSI C," *OOPS Messenger*, Vol. 2, No. 3, pp. 12–23, July 1991.

[OOPSLA 1987] *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '87) Proceedings*, October 1987.

[OOPSLA 1988] *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '88) Proceedings*, September 1988.

[OOPSLA 1989] *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings*, October 1989.

[Par 1992] ParcPlace Systems: *Objectkit\Smalltalk C Programming*, 1992.

[Strong *et al.* 1958] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel: "The Problem of Programming Communication with Changing Machines: A Proposed Solution," *Communications of the ACM*, Vol. 1, No. 8, pp. 12–18, August 1958. (Part 2: Vol. 1, No. 9, pp. 9–15).

[Stroustrup 1986] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley. 1986.

[Sun 1985] Sun Microsystems: *External Data Representation Reference Manual*, 1985.

[Suzuki 1981] N. Suzuki: "Inferring Types in Smalltalk," in *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 187–199, January 1981.

[Ungar and Jackson 1988] D. Ungar and F. Jackson: "Tenuring Policies for Generation-Based Storage Reclamation," in OOPSLA [OOPSLA 1988], pp. 1–17.

[Ungar and Jackson 1992] D. Ungar and F. Jackson: "An Adaptive Tenuring Policies for Generation Scavengers," *ACM Transactions on Programming Languages and Systems*, Vol. 14, No. 1, pp. 1–27, January 1992.

[Ungar and Smith 1987] D. Ungar and R. B. Smith: "SELF: The Power of Simplicity," in OOPSLA [OOPSLA 1987], pp. 227–241.

[Ungar 1984] D. Ungar: "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 157–167, April 1984.

[Vokach-Brodsky and Wolczko 1990] B. R. Vokach-Brodsky and M. I. Wolczko: "Smalltalk Application Compilers," in *Proceedings of TOOLS 3*, pp. 69–78, 1990.

[Weiser *et al.* 1989] M. Weiser, A. Demers, and C. Hauser: "The Portable Common Runtime Approach to Interoperability," in *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pp. 114–122, December 1989.

[Wirfs-Brock and Wilkerson 1988] A. Wirfs-Brock and B. Wilkerson: "An Overview of Modular Smalltalk," in OOPSLA [OOPSLA 1988], pp. 123–134.

[Yuasa and Hagiya 1985] T. Yuasa and M. Hagiya: *Kyoto Common Lisp Report*. Teikoku Insatsu, 1985.

Appendix A

Algorithm of Garbage Collection

A.1 Algorithm of the Collector I

We present the algorithm of the Collector I in the C language:

```
struct space {
  word_t *firstWord; /* first word of space */
  word_t *boundary;  /* boundary of space */
  int    size;       /* number of used words in space */
};

typedef struct otEntry {
  word_t  *contents;
  struct {
    unsigned isNew       :1;
    unsigned isForwarded :1;
    unsigned isRemembered:1;
    unsigned             :5;
    unsigned age         :8;
    unsigned size        :16;
  } gcInfo;
} *oop;

struct otEntry OT[MaxObject];
struct space
        CreationSpace,
        PastSSpace,   /* PastSurvivorSpace */
        FutureSSpace, /* FutureSurvivorSpace */
        OldSpace;
oop RemSet[MaxRemembered]; /* RememberedSet */
int RemSetSize;
}

/*
```

The main routine. generationScavengeWithAmbiguousRoots(). first scavenges the new objects immediately reachable from ambiguous roots.

Next, it scavenges the new objects immediately reachable from old ones. Then it scavenges those that are transitively reachable. If this results in a tenuring, the tenuree gets remembered. and it first scavenges objects adjacent to the tenuree, then scavenges the ones reachable from the tenured. This loop continues until no more reachable objects are left.

At that point, PastSSpace is exchanged with FutureSSpace and free OT entries are reclaimed.
*/

```
generationScavengeWithAmbiguousRoots()
{
  int prevRemSetSize;
  int prevFutureSSpaceSize;

  scavengeAmbiguousRoots();

  prevRemSetSize = 0;
  prevFutureSSpaceSize = 0;

  while (TRUE) {
    scavengeRemSetStartingAt(prevRemSetSize);
    if (prevFutureSSpaceSize == FutureSSpace.size)
      break;

    prevRemSetSize = RemSetSize;
    scavengeFutureSSpaceStartingAt(
                             prevFutureSSpaceSize);
    if (prevRemSetSize == RemSetSize)
      break;

    prevFutureSSpaceSize = FutureSSpace.size;
  }

  exchange(PastSSpace, FutureSSpace);

  reclaimOTEntries();
}
```

/*

scavengeAmbiguousRoots() inspects all the words in ambiguous roots (the stack. registers. and the data area) by using conservative OOP-finding. If the word refers to any valid new OT entries. its new referents are scavenged.
*/

```
scavengeAmbiguousRoots()
{
  word_t sp;
  int reg;
  oop value;

  for (sp = Stack_bottom;
       sp <= Satck_top;
       sp += FIND_INCREMENT)
  {
    value = (oop)*sp;
    if (conservativePointerFinding(value))
      if (! value->gcInfo.isForwarded)
```

```
        copyAndForwardObject(value);
  }
  for (reg = Register_first;
       reg <= Register_last;
       reg += 1)
  {
    value = (oop)processorRegister(reg);
    if (conservativePointerFinding(value))
      if (! value->gcInfo.isForwarded)
        copyAndForwardObject(value);
  }
  for (sp = Area_bottom;
       sp <= Area_top;
       sp += FIND_INCREMENT)
  {
    value = (oop)*sp;
    if (conservativePointerFinding(value))
      if (! value->gcInfo.isForwarded)
        copyAndForwardObject(value);
  }
}
```

/*

`scavengeRemSetStartingAt(dest)` traverses objects in the remembered set starting at the `dest`-th one. If the object does not refer to any new objects, it is removed from the set. Otherwise, its new referents are scavenged.

*/

```
scavengeRemSetStartingAt(dest)
int dest;
{
  int src;

  for (src = dest; src < RemSetSize; ++src)
    if (scavengeReferentsOf(RemSet[src])) {
      RemSet[dest++] = RemSet[src];
    }
    else
      RemSet[src]->gcInfo.isRem = FALSE;
  RemSetSize = dest;
}
```

/*

`scavengeFutureSSpaceStartingAt(n)` inspects words starting at the `n`-th word of `FutureSSpace` by using `conservative OOP-finding`. If the word refers to any valid new OT entries, its new referents are scavenged.

*/

```
scavengeFutureSSpaceStartingAt(n)
int n;
{
  oop value;

  for (;
    n < FutureSSpace.size;
    n ++)
```

```
  {
    value = (oop)FutureSSpace.firstWord[n];
    if (conservativePointerFinding(value))
      if (! value->gcInfo.isForwarded)
        copyAndForwardObject(value);
  }
}
```

/*
scavengeReferentsOf(obj) inspects all the words in obj by using conservative OOP-finding. If the word refers to any valid new OT entries. its new referents are scavenged. and returns truth. If there are no new referents. it returns falsity.
*/

```
scavengeReferentsOf(obj)
oop obj;
{
  int i;
  int foundNewReferent;
  oop value;

  foundNewReferent = FALSE;
  for (i = 0; i < obj->gcInfo.size; i++) {
    value = (oop)obj->contents[i];
    if (conservativePointerFinding(value)) {
      foundNewReferent = TRUE;
      if (! value->gcInfo.isForwarded)
        copyAndForwardObject(value);
    }
  }
  return foundNewReferent;
}
```

/*
copyAndForwardObject(obj) copies a new object either to FutureSSpace. or if it is to be tenured. to OldSpace. It leaves a forwarding pointer behind.
*/

```
copyAndForwardObject(obj)
oop obj;
{
  word_t *newLocation;

  if (obj->gcInfo.age < MaxAge) {
    ++ obj->gcInfo.age;
    newLocation = copyObjectToSpace(
                                 obj->contents,
                                 obj->gcInfo.size,
                                 FutureSSpace);
    obj->gcInfo.isForwarded = TRUE;
  }
  else {
    newLocation = copyObjectToSpace(
                                 obj->contents,
                                 obj->gcInfo.size,
                                 OldSpace);
```

```
    obj->gcInfo.isNew = FALSE;
    RemSet[RemSetSize++] = obj;
    obj->gcInfo.isRemembered = TRUE;
  }
  obj->contents = newLocation;
}
```

/*
reclaimOTEntries() traverses all new entries in OT. If the entry is forwarded, its isForwarded flag is reset to false. Otherwise, it is reclaimed into free OT list.
*/

```
reclaimOTEntries();
{
  int i;

  for (i = 0; i < MaxObject; i++) {
    if (OT[i].gcInfo.isNew)
      if (OT[i].gcInfo.isForwarded)
        OT[i].gcInfo.isForwarded = FALSE;
      else
        addFreeOTList(OT[i]);
  }
}
```

A.2 Algorithm of the Collector II

We present the algorithm of the Collector II in the C language:

```
struct space {
  word_t *firstWord; /* first word of space */
  word_t *boundary;  /* boundary of space */
  int    size;       /* number of used words in space */
};

typedef struct otEntry {
  word_t  *contents;
  struct {
    unsigned isNew        :1;
    unsigned isForwarded  :1;
    unsigned isRemembered :1;
    unsigned isOTForwarded:1;
    unsigned              :4;
    unsigned age          :8;
    unsigned size         :16;
  } gcInfo;
  struct objInfo_t objInfo;
} *oop;

struct otEntry NewOt[MaxNewObject],
               OldOT[MaxOldObject];
```

```
struct space
        CreationSpace,
        PastSSpace,   /* PastSurvivorSpace */
        FutureSSpace, /* FutureSurvivorSpace */
        OldSpace;
oop RemSet[MaxRemembered]; /* RememberedSet */
int RemSetSize;
```

/*

The main routine. `generationScavengeWithAmbiguousRoots()`. first scavenges the new objects immediately reachable from ambiguous roots.

Next. it scavenges the new objects immediately reachable from old ones. Then it scavenges those that are transitively reachable. If this results in a tenuring, the tenuree gets remembered. and it first scavenges objects adjacent to the tenuree, then scavenges the ones reachable from the tenured. This loop continues until no more reachable objects are left.

At that point, `PastSSpace` is exchanged with `FutureSSpace` and free `NewOT` entries are reclaimed.

*/

```
generationScavengeWithAmbiguousRoots()
{
  int prevRemSetSize;
  int prevFutureSSpaceSize;

  scavengeAmbiguousRoots();

  prevRemSetSize = 0;
  prevFutureSSpaceSize = 0;

  while (TRUE) {
    scavengeRemSetStartingAt(prevRemSetSize);
    if (prevFutureSSpaceSize == FutureSSpace.size)
      break;

    prevRemSetSize = RemSetSize;
    scavengeFutureSSpaceStartingAt(
                        prevFutureSSpaceSize);
    if (prevRemSetSize == RemSetSize)
      break;

    prevFutureSSpaceSize = FutureSSpace.size;
  }

  exchange(PastSSpace, FutureSSpace);

  reclaimNewOTEntries();
}
```

/*

`scavengeAmbiguousRoots()` inspects all the words in ambiguous roots (the stack, registers. and the data area) by using `conservative OOP-finding`. If the word refers to any valid new OT entries, its new referents are scavenged.

*/

```
scavengeAmbiguousRoots()
{
  word_t sp;
```

```
  int reg;
  oop value;

  for (sp = Stack_bottom;
       sp <= Satck_top;
       sp += FIND_INCREMENT)
  {
    value = (oop)*sp;
    if (conservativePointerFinding(value))
      if (! value->gcInfo.isForwarded)
        copyAndForwardObject(value, TRUE);
  }
  for (reg = Register_first;
       reg <= Register_last;
       reg += 1)
  {
    value = (oop)processorRegister(reg);
    if (conservativePointerFinding(value))
      if (! value->gcInfo.isForwarded)
        copyAndForwardObject(value, TRUE);
  }
  for (sp = Area_bottom;
       sp <= Area_top;
       sp += FIND_INCREMENT)
  {
    value = (oop)*sp;
    if (conservativePointerFinding(value))
      if (! value->gcInfo.isForwarded)
        copyAndForwardObject(value, TRUE);
  }
}
```

/*
scavengeRemSetStartingAt(dest) traverses objects in the remembered set starting at the dest-th one. If the object does not refer to any new objects, it is removed from the set. Otherwise, its new referents are scavenged.
*/

```
scavengeRemSetStartingAt(dest)
int dest;
{
  int src;

  for (src = dest; src < RemSetSize; ++src)
    if (scavengeReferentsOf(RemSet[src])) {
      RemSet[dest++] = RemSet[src];
    }
    else
      RemSet[src]->gcInfo.isRemembered = FALSE;
  RemSetSize = dest;
}
```

/*
scavengeFutureSSpaceStartingAt(n) traverses objects starting at the n-th word of FutureSSpace. At this point, a reversed pointer is forwarded.

For simplicity here, an object is just an array of pointers, so the `objInfo` field in the OT entry is not used.
*/

```
scavengeFutureSSpaceStartingAt(n)
int n;
{
  oop obj;
  int dontCare;

  for (;
    n < FutureSSpace.size;
    n += obj->gcInfo.size)
  {
    /* Forward a reversed pointer */
    obj = FutureSSpace.firstWord[n];
    FutureSSpace.firstWord[n] = obj->contents;
    obj->contents = &FutureSSpace.firstWord[n];
    dontCare = scavengeReferentsOf(obj);
  }
}
```

/*
`scavengeReferentsOf(obj)` inspects all the pointers in `obj`. If any are new objects, its new referents are scavenged, and returns truth. If there are no new referents, it returns falsity.

For simplicity here, an object is just an array of pointers, so the `objInfo` field in the OT entry is not used.
*/

```
scavengeReferentsOf(obj)
oop obj;
{
  int i;
  int foundNewReferent;
  oop referent;

  foundNewReferent = FALSE;
  for (i = 0; i < obj->gcInfo.size; i++) {
    referent = (oop)obj->contents[i];
    if (referent->gcInfo.isNew) {
      foundNewReferent = TRUE;
      if (! referent->gcInfo.isForwarded) {
        if (! referent->gcInfo.isOTForwarded)
          copyAndForwardObject(referent, FALSE);
        if (referent->gcInfo.isOTForwarded)
          obj->contents[i] =
            (word_t)referent->contents;
      }
    }
  }
  return foundNewReferent;
}
```

/*

copyAndForwardObject(obj, isLocked) copies a new object either to FutureSSpace. or if it is to be tenured. to OldSpace.

If obj is not to be tenured. a pointer from obj to its new location is reversed. Otherwise. if isLocked is false, its OT entry is copied into OldOT and a forwarding OT pointer is left.
*/

```
copyAndForwardObject(obj, isLocked)
oop obj;
int isLocked
{
  word_t *newLocation;
  oop newOTLocation;

  if (obj->gcInfo.age < MaxAge) {
    ++ obj->gcInfo.age;
    newLocation = copyObjectToSpace(
                             obj->contents,
                             obj->gcInfo.size,
                             FutureSSpace);
    /* Reverse a pointer */
    obj->contents = newLocation[0];
    newLocation[0] = obj;
    obj->gcInfo.isForwarded = TRUE;
  }
  else {
    newLocation = copyObjectToSpace(
                             obj->contents,
                             obj->gcInfo.size,
                             OldSpace);
    if (isLocked) {
      obj->contents = newLocation;
      obj->gcInfo.isNew = FALSE;
      RemSet[RemSetSize++] = obj;
      obj->gcInfo.isRemembered = TRUE;
    }
    else {
      newOTLocation = copyOTEntryToOT(obj, OldOT);
      newOTLocation->contents = newLocation;
      newOTLocation->gcInfo.isNew = FALSE;
      obj->contents = newOTLocation;
      obj->gcInfo.isOTForwarded = TRUE;
      RemSet[RemSetSize++] = newOTLocation;
    }
  }
}
```

/*

reclaimNewOTEntries() traverses all entries in NewOT. If the entry is forwarded. its isForwarded flag is reset to false. Otherwise. it is reclaimed into free NewOT list.
*/

```
reclaimNewOTEntries();
{
  int i;

  for (i = 0; i < MaxNewObject; i++) {
    if (NewOT[i].gcInfo.isNew)
```

```
      if (NewOT[i].gcInfo.isForwarded)
        NewOT[i].gcInfo.isForwarded = FALSE;
      else
        addFreeNewOTList(NewOT[i]);
    }
  }
```