Chapter 1   Introduction

1.1   Introduction

Programming language designers have always been searching for programming languages and features that ease the programming process and improve programmer productivity. One promising current approach is *object-oriented programming*. Object-oriented programming languages provide programmers with powerful techniques for writing, extending, and reusing programs quickly and easily. Object-oriented languages typically involve some sort of *inheritance* of implementation, allowing programmers to implement new abstract data types in terms of implementations of existing abstract data types, and some sort of *message passing* to invoke operations on objects of unknown implementation. Several object-oriented languages have been designed and implemented, including Smalltalk-80* [GR83], C++ [Str86, ES90], Trellis/Owl [SCW85, SCB+86], Eiffel [Mey86, Mey88, Mey92], Modula-3 [Nel91, Har92], CLOS [BDG+88], and T [RA82, Sla87, RAM90]. Unfortunately, traditional implementations of object-oriented language features, particularly message passing, have been much slower than traditional implementations of their non-object-oriented counterparts, and this gap in run-time performance has limited the widespread use of object-oriented language features and hindered the acceptance of purely object-oriented languages.

Language designers have developed several other important language and environment features to improve programmer productivity. First-class *closures* allow programmers to define their own control structures such as iterators over collection-style abstract data types and exception handling routines. *Generic arithmetic* supports general numeric computation over a variety of numeric representations without explicit programmer intervention. A *safe*, *robust* implementation performs all necessary error checking to ensure that programs do not behave in an implementation-dependent way (e.g., get a "mystery core dump") when they contain errors such as array access out of bounds or stack overflow. Complete *source-level debugging* helps programmers get programs working quickly, significantly improving programmer productivity. Unfortunately, most languages and implementations do not support all these desirable features, again because they historically have had a high cost in run-time performance.

1.2   The SELF Language

To maximize the potential benefits of object-oriented programming, David Ungar and Randy Smith designed the SELF programming language [US87, HCC+91, UCCH91, CUCH91] as a refinement and simplification of the Smalltalk-80 language. SELF incorporates a purely object-oriented programming model, closures for user-defined control structures, generic arithmetic support, a safe, robust language implementation, and support for complete source-level debugging. (SELF will be described in more detail in Chapter 4.) Ungar and Smith strove to provide a simple, flexible language and environment that maximized the expressive power and productivity of the programmer. However, SELF's powerful features initially appeared to make its implementation prohibitively inefficient: the fastest implementation of Smalltalk-80, a language that does not include all the features of SELF, runs a set of small benchmark programs at only a tenth the speed of optimized C programs.

1.3   Our Research

The goal of the work described in this dissertation is to design and build an efficient implementation of SELF on stock hardware that does not sacrifice any of the advantages of the language or environment. Achieving our goal required us to develop new implementation strategies for message passing, closures and user-defined control structures, generic arithmetic, robust primitives, and source-level debugging. Our results have been surprisingly good: the same set of benchmarks used to measure the performance of Smalltalk-80 programs indicate that SELF programs run between a third and half the speed of the optimized C programs, roughly five times faster than the Smalltalk-80 implementation. These new techniques are practical, since SELF's compilation speed is roughly the same as an optimizing C compiler, and SELF's compiled code space usage is usually within a factor of two of optimized C.

Our new implementation strategies work well in overcoming the obstacles to good performance for SELF. Fortunately, these new techniques could be included in the implementations of other object-oriented languages to improve their

* Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

run-time performance. Even non-object-oriented languages incorporating user-defined control structures and/or generic arithmetic could benefit from our new implementation techniques. More importantly, we hope that the techniques we have developed will pave the way for other purely object-oriented languages to be designed and implemented without including compromises or restrictions solely for the sake of efficiency.

1.4 Outline

The next chapter of this dissertation describes the benefits of object-oriented programming, user-defined control structures, generic arithmetic support, and a robust implementation, as well as their associated costs in run-time performance. It also outlines various compromises and restrictions that other languages have included to achieve better run-time efficiency. Chapter 3 reviews this related work in detail. Chapter 4 describes the SELF language.

Chapters 5 through 13 contain the meat of the dissertation. Chapter 5 presents the goals of this work and outlines the organization of the compiler. Chapter 6 describes the framework in which the compiler functions, including the memory system architecture and the run-time system. (An early design and implementation of the memory system was described in Elgin Lee's thesis [Lee88].) Chapters 7 through 12 present the bulk of the new techniques developed to improve run-time performance. These techniques include *customization*, *type analysis*, *type prediction*, and *splitting*. Earlier designs and implementations of these techniques have been described in other papers [CU89, CUL89, CU90, CU91]. Chapter 13 describes compiler support for the SELF programming environment, in particular techniques that mask the effects of optimizations such as inlining and splitting from the SELF programmer when debugging; some of these techniques have been described in other papers [CUL89, HCU92]. Section 5.3 contains a more detailed outline of this part of the dissertation.

The performance of our SELF implementation is analyzed in Chapter 14. This analysis measures various configurations of our implementation and identifies the individual contributions to performance of particular techniques. The compile time and space costs of the system as a whole and of individual techniques are analyzed as well.

Finally, Chapter 15 concludes the dissertation and outlines some areas for future work.