Chapter 5 Overview of the Compiler

This chapter introduces the overall design of the SELF compiler. The next section describes the goals of our work. Section 5.2 describes the primary purpose of most of the new techniques developed as part of the SELF compiler: the extraction of representation-level type information. Section 5.3 relates the novel parts of the SELF compiler to the traditional front-end and back-end division of a compiler and outlines topics covered by the next several chapters.

5.1 Goals of Our Work

Our immediate goal is to build an efficient, usable implementation of SELF on stock hardware. However, we are not willing to compromise SELF's pure object model and other expressive features. We want to preserve the illusion of the system directly executing the program as the programmer wrote it, with no user-visible optimizations. This constraint has several consequences that distinguish our work from other optimizing language implementations:

- The programmer must be free to edit any procedure in the system, including the most basic ones such as the definition of **+** for integers and **`ifTrue:`** for booleans, and provide overriding definitions for other data types when desired.
- The programmer must be able to understand the execution of the program and any errors in the program solely in terms of the source code and the source language constructs. This requirement on the debugging and monitoring interface to the system disallows any internal optimizations that would shatter the illusion of the implementation directly executing the source program as written. Programmers should be unaware of how their programs get compiled or optimized.
- The programmer should be isolated even from the mere fact that the programs are getting compiled at all. No explicit commands to compile a method or program should ever be given, even after programming changes. The programmer just runs the program.

  This illusion of hiding the compiler would break down if the programmer were distracted by mysterious pauses due to compilation, analysis, or optimization. Ideally any pauses incurred by the implementation of the system would be imperceptible, such as on the order of a fraction of a second when in interactive use. Longer running batch programs can be interrupted by longer pauses, as long as the total time of the program is not slowed so much that the programmer becomes aware of the pauses.

Within these constraints on the user-visible semantics of the system, our main objective is excellent run-time performance. We wish to make SELF and other pure object-oriented languages with similar powerful features competitive in performance with traditional non-object-oriented languages such as C and Pascal. In particular, where the SELF program is not taking advantage of object-oriented features, such as in an inner loop that could have been written just as easily in C as in SELF, we want the performance of SELF to be close to the performance of optimized C. If these performance goals are met, many programmers may be able to switch from traditional languages to pure object-oriented languages and begin to reap the benefits afforded by pure object-oriented programming, user-defined control structures, generic arithmetic, and robust primitives.

Other goals are secondary to the constraint of a source-level execution model and the goal of rapid execution. In particular, run-time and compile-time space overheads are less of a concern than run-time speed. Modern computer platforms, especially workstations, are typically equipped with a large amount of physical main memory, and this amount is increasing at a rapid rate. We therefore are willing to use more space than would a straightforward implementation in order to meet our execution speed goals.

When we began this work, there were no techniques available to implement pure object-oriented languages like SELF efficiently without relying on special-purpose hardware support, cheating in the implementation, or diluting the object-oriented model by introducing non-object-oriented constructs into the language. Therefore the main part of our work involved developing and implementing new techniques for implementing object-oriented languages efficiently on stock hardware. These new techniques have enabled us to largely meet our goals both for faithfulness to the source code and run-time execution speed.

Of course, we do not only wish to implement SELF efficiently, but also a larger class of SELF-like languages. Fortunately, the new techniques are not specific to the SELF language. Most object-oriented languages, including C++, Eiffel, Trellis/Owl, Smalltalk, T, and CLOS, would benefit (to varying degrees) from the techniques we have developed. Also, languages with object-oriented subsystems would benefit, including languages supporting some form of generic arithmetic such as Lisp, APL [Ive62, GR84], PostScript [Ado85], and Icon [GG83], languages with logic variables such as Prolog, and languages with futures such as Multilisp [Hal85] and Mul-T [KHM89].

5.2 Overall Approach

This section describes the overall approach to achieving an efficient implementation of SELF and similar languages.

5.2.1 Representation-Level Type Information Is Key

Dynamically-typed object-oriented programming languages historically have run much slower than traditional statically-typed non-object-oriented programming languages. This performance gap is attributable largely to the lack of representation-level type information in the dynamically-typed object-oriented languages. This representation-level information about an object is embodied in the object's *class* in a class-based system. (Section 6.1.1 will describe *maps*, internal implementation structures that embody representation-level type information for prototype-based languages such as SELF.)

If the compiler could infer the classes (or maps) of objects at compile-time, it could eliminate much of the run-time overhead associated with dynamic typing and object orientation. In a dynamically-typed language, the compiler must insert extra run-time type-checking code around type-safe primitives and extra run-time type-casing code in support of generic arithmetic. If the compiler could infer the classes of the arguments to the type-checking primitive, then it could perform the type checks at compile-time rather than run-time. Similarly, if the compiler could infer the classes of the arguments to generic arithmetic primitives, then it could perform the type-casing at compile-time, generating code for a type-specific arithmetic operation instead of the slower generic operation.

In an object-oriented language, the compiler must insert extra run-time message dispatching code to implement dynamic binding of message names to target methods based on the run-time class of the receiver. If the compiler could infer the class of the receiver of a message, then it could perform message lookup at compile-time instead of run-time, replacing the dynamically-bound message with a statically-bound procedure call; the statically-bound call would subsequently be amenable to further optimizations such as inlining (described in Chapter 7) that can significantly boost performance.

5.2.2 Interface-Level Type Declarations Do Not Help

Clearly, the run-time performance of dynamically-typed object-oriented programs could be dramatically improved if the compiler could infer representation-level type information in the form of objects' classes or maps. On the surface, this would seem to imply that statically-typed object-oriented languages, with lots of type information available to the compiler, would have a huge advantage in performance over their dynamically-typed counterparts. Perhaps surprisingly, this advantage is in fact quite small.

In a non-object-oriented language, the type of a variable specifies the representation or implementation of the contents of the variable. This static information corresponds to knowing the exact class of the contents of the variable and hence supports the optimizations described above that reduce the gap between dynamically-typed object-oriented languages and statically-typed non-object-oriented languages. In an object-oriented language with interface-level type declarations, however, the type of a variable specifies only the set of operations that are guaranteed to be implemented by objects stored in the variable. The interface-level type deliberately does *not* specify anything about how the objects stored in the variable will *implement* the operations, in order to maximize the generality and reusability of the code. With only interface-level type information, the compiler cannot perform the optimizations that require representation-level type information. For example, knowing that an object *understands* the **+** message does not help the compiler generate more efficient code for the **+** message; only knowledge about how the object *implements* the **+** message (such as by executing the **+** method for integers) enables optimizations such as inlining that markedly improve performance.[*]

[*] Interface-level type information can be useful in special cases given system-wide knowledge. The compiler can examine all the possible implementations in the system that satisfy some interface and sometimes infer useful representation-level type information. For example, if only one object or class implements a particular interface, the static interface-level type information implies representation-level information. These kinds of optimizations are less likely to speed operations on basic data structures such as numbers and collections, however, where many implementations of the same interface are the norm.

5.2.3 Transforming Polymorphic into Monomorphic Code

The lack of static representation-level type information limits the run-time performance of object-oriented languages, whether dynamically-typed or statically-typed. Consequently, our new compilation techniques will strive to infer this missing representation-level type information, so that the compiler can perform optimizations to eliminate the overhead of dynamic typing and object orientation. Once these optimizations have been performed, the task of compiling a dynamically-typed object-oriented program reduces to the task of compiling a traditional statically-typed procedural program.

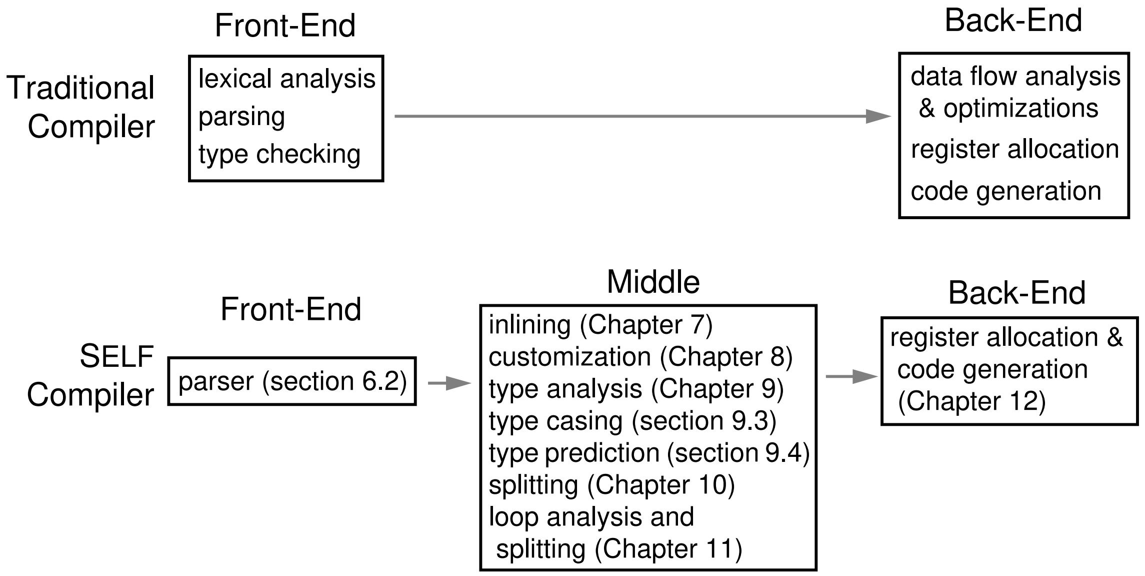
To perform inlining the SELF compiler must prove that the receiver of a message has a single representation, i.e., that the receiver expression is *monomorphic*. In general, however, SELF code is *polymorphic*: expressions may denote values of different representations at different times, and the same source code works fine for all these representations. Such polymorphism is central to the power of object-oriented programming. However, in some cases the SELF program does not exploit the full power of polymorphism. Sometimes a message receiver can be a member of only a single clone family. To exploit such cases, the compiler includes techniques such as *type analysis* (described in Chapter 9) to identify monomorphic expressions and subsequently optimize them.

In most cases, however, the compiler's task is not so easy: most SELF expressions really are potentially polymorphic. Nevertheless, the compiler can frequently optimize even polymorphic messages. The compiler includes several techniques such as *customization* (described in Chapter 8), *type casing* (described in section 9.3), *type prediction* (described in section 9.4), and *splitting* (described in Chapter 10) that can transform some kinds of polymorphic expressions into monomorphic expressions; these monomorphic expressions are then suitable for further optimization. All these techniques work by duplicating code, transforming a single polymorphic expression with *N* possible representations into *N* separate monomorphic expressions. Each monomorphic case can be optimized independently; without this separation no optimization would be possible. These techniques for trading away compiled-code space to gain run-time speed form the heart of the SELF compiler and are our key contribution to compilation technology for object-oriented languages.

Since identifying and creating monomorphic sections of code can be fairly time consuming, the SELF compiler seeks to conserve its efforts. In particular, the compiler attempts to compile only those parts of the SELF program that are actually executed. The compiler only performs customization on demand, exploiting SELF's dynamic compilation architecture as described in section 8.2. Additionally, many cases that could arise in principle but rarely arise in practice, such as integer overflows, array accesses out of bounds, or illegally-typed arguments to primitives, are never actually compiled by the SELF compiler, thus saving a lot of compile time and compiled code space and allowing better optimization of the parts of programs that *are* executed. This *lazy compilation of uncommon branches* is described in section 10.5.

5.3 Organization of the Compiler

Traditional compilers are typically divided into a *front-end*, which performs lexical analysis and parsing, and a *back-end*, which performs optimizations and generates code. The SELF parser, described in section 6.2, performs the functions of a traditional front-end by translating SELF source into a byte-coded representation. The SELF compiler performs many of the functions of a traditional back-end; Chapter 12 describes the SELF compiler's version of most of these traditional functions. The bulk of the SELF compiler effort, however, lies in between the two halves of a traditional compiler. This "middle half" of the SELF compiler performs the representation-level type analysis and inlining that bridges the semantic gap between the high-level polymorphic program input to the SELF compiler and the lower-level monomorphic version of the program suitable for the optimizations performed by a traditional compiler back-end.


Front-End
Back-End
Traditional Compiler
lexical analysis
parsing
type checking
data flow analysis & optimizations
register allocation
code generation
Middle
Front-End
Back-End
SELF Compiler
parser (section 6.2)
inlining (Chapter 7)
customization (Chapter 8)
type analysis (Chapter 9)
type casing (section 9.3)
type prediction (section 9.4)
splitting (Chapter 10)
loop analysis and splitting (Chapter 11)
register allocation & code generation (Chapter 12)

The next chapter describes the supporting services provided by the rest of the SELF system architecture, including a description of the *map* data structures that convey representation-level type information of objects to the compiler. The "middle half" of the SELF compiler is described in the following several chapters. Chapter 7 describes inlining in more detail. Chapter 8 presents *customization*, one the SELF compiler's important new techniques. Chapter 9 describes *type analysis*, the technique used by the SELF compiler to infer and propagate the representation-level type information through the control flow graph. It also presents *type prediction*, a technique for guessing the types of some objects based on the names of messages and built-in profile information. Chapter 10 describes *splitting*, the primary technique used in the SELF compiler to turn polymorphic pieces of code into multiple monomorphic pieces of code. Chapter 10 also describes *lazy compilation*, a technique for only compiling those parts of methods that the compiler judges to be likely to be executed. Chapter 11 concludes the discussion of the middle end by describing type analysis and splitting in the presence of loops.