Chapter 15 Conclusions

15.1 Results of this Work

With this work we have striven to demonstrate that pure object-oriented languages can be efficient on stock hardware, given suitable implementation techniques. To achieve high efficiency, we had to design and implement several new implementation techniques, including *customization*, *type prediction*, *iterative type analysis*, and *splitting*. Customization and type prediction extract representation-level type information from untyped source programs, and type analysis and splitting preserve this valuable information as long as possible. The accumulated type information then is used to statically-bind and inline away messages, especially those involved in user-defined control structures and generic arithmetic, leading to dramatic performance improvements.

By lazily compiling uncommon cases such as arithmetic overflows and primitive failures, the compiler can concentrate its efforts on the common-case parts of a program while still supporting the uncommon events if they should occur. This strategy resolves the tension between fast execution and powerful language features by providing the best of both worlds: good execution and compilation speed for common cases and support for more powerful but less common cases.

With these techniques the current SELF implementation runs small- to medium-sized benchmarks at about half the speed of optimized C, with compile times that are comparable to the optimizing C compiler and compiled code space usage that is less than double that of the optimizing C compiler. This new-found execution speed is more than five times faster than the fastest previous implementation of a similar language, ParcPlace Smalltalk-80.

Several general themes underlie this work. Our techniques frequently trade away space for speed, compiling multiple specialized versions of a single piece of source code; customization and splitting exemplify this approach. To minimize the compile time and compiled code space costs of this approach, many of our techniques are applied *lazily*. Methods are compiled and specialized lazily, only when first invoked; uncommon parts of the control flow graph are compiled lazily, only when first taken. Lazy compilation appears to be the saving grace which makes specialization practical.

15.2 Applicability of the Techniques

The techniques developed for SELF optimize programs that make heavy use of message passing. These techniques also apply to other languages that share these properties. Clearly, other pure, dynamically-typed object-oriented languages such as Smalltalk-80 could benefit directly from these techniques. As discussed in section 5.2.2, our techniques also apply to relatively pure, statically-typed object-oriented languages such as Trellis/Owl and Eiffel. Hybrid languages such as C++ and many object-oriented Lisps have less need for our techniques, since performance-critical parts of programs can be written in the lower-level non-object-oriented subset of the language. However, our techniques would still be useful to the extent that implementations wish to support and encourage the use of the object-oriented features of their languages. One researcher already has proposed extending C++ to support a form of customization [Lea90].

Our techniques also could improve the performance of many languages that do not claim to be object-oriented. These languages include powerful features in which several different representations of objects can be used interchangeably within programs. This ability is essentially the same as message passing, except that the set of possible representations usually is not user-extensible, and so we argue that these languages contain object-oriented subsets. Our techniques would be useful in improving performance of these languages to the extent that these “object-oriented” features are used by programs. For example, non-object-oriented languages supporting generic arithmetic, such as most Lisps and Icon, could significantly benefit from the inclusion of type analysis, type prediction, splitting, and lazy compilation of uncommon cases to extract and preserve representation-level information for optimization. To illustrate, our SELF implementation generates code for benchmarks using generic arithmetic that runs more than twice as fast as the code generated by the ORBIT compiler for the T dialect of Scheme, even though the T benchmarks use no message passing or user-defined control structures. Even when the T version of the benchmarks is rewritten to use unsafe integer-specific arithmetic (giving up the semantics of generic arithmetic), SELF still runs faster. Based on this result, we argue that language designers, implementors, and users should abandon unsafe integer-specific arithmetic in favor of safe, expressive generic arithmetic combined with optimization techniques like ours.

Language features other than generic arithmetic might benefit from our techniques. APL allows programs to manipulate scalars, arrays, and matrices of arbitrary dimension interchangeably, and our techniques might be used to lazily compile dimension-specific code to speed APL programs. Implementations of logic programming languages such as Prolog might benefit from knowing that along certain branches some logic variable is guaranteed to be instantiated; this knowledge could come from techniques related to type analysis and splitting. Similarly, implementations of programming languages supporting *futures* such as Multilisp [Hal85] and Mul-T [KHM89] could distinguish between known and unknown futures, compiling specialized code for each case (or perhaps just for the common case of known futures). Thus, our techniques may be more broadly applicable to a variety of modern programming languages beyond only pure object-oriented languages.

15.3 Future Work

While significant progress has been made in moving SELF and other pure object-oriented programming languages into the realm of practicality, more work remains to complete the task. Some applications require the maximum in efficiency, such as scientific and numerical applications like those traditionally written in Fortran. SELF as currently implemented is probably not efficient enough for such demanding users. One avenue of future research therefore would push the upper limits of performance towards that achieved for traditional languages and to extend the current implementation techniques to handle floating point representations as efficiently as integer representations are currently handled in the SELF implementation.

A related direction would attempt to validate that these techniques scale to much larger systems than have been measured so far. Several of our techniques rely on trading away compiled code space for run-time speed. For the systems measured, in the 100- to 1000-line program range, this potential space explosion has not been a problem in practice, but for larger programs, on the order of 10,000 or 100,000 lines, the concern still remains. More research could be done to ensure that the techniques are robust in the face of such large systems.

A third direction would focus on further improving the performance of object-oriented programs. Only a few of the benchmarks measured so far make significant use of the extra power of the SELF language beyond what is available in traditional languages. The question remains of how well our techniques will fare for programs that make heavy use of the advanced features of the language. Ideally, object-oriented programs would be written much faster and would be easier to change and extend than equivalent non-object-oriented programs, and would run just as fast as the non-object-oriented versions. This goal is not yet met by the current SELF implementation, which for the **`richards`** benchmark runs about a third the speed of the optimized C version. Some initial work has already begun in this direction [HCU91].

A final direction would address more of the programming environment issues. While the current SELF compiler compiles as fast as the optimizing C compiler on small- and medium-sized benchmarks (and compiles more than twice as fast as an optimizing C++ compiler), the fact that compilation takes place at run-time for SELF holds our system up to a higher standard. Users tend to become distracted by pauses of more than a fraction of a second, either from garbage collection or from run-time compilation, and their productivity drops correspondingly. Pauses of more than a dozen seconds or so bring about an even more severe distraction and decline in productivity. The current SELF implementation might meet the second level of performance but unfortunately is still not at the level of fraction-of-a-second compile pauses. To maintain a high-productivity environment, more research is needed to reconcile unnoticeable compiler pauses with good run-time performance. Fortunately, this problem also is being actively pursued [HCU91] and early results are quite promising.

15.4 Conclusion

We believe that this work has demonstrated the feasibility of the new techniques and consequently the practicality of pure object-oriented languages for a wide range of applications. We hope that this demonstration will convince future language designers to avoid compromises in their designs motivated solely by concerns over the efficiency of a pure message passing model. We also hope that language users will begin to demand such simple, flexible languages.