RETROSPECTIVE:

TIL: A Type-Directed, Optimizing Compiler for ML

David Tarditi
Microsoft Research

Greg Morrisett
Cornell University

Perry Cheng
IBM Research

Chris Stone
Harvey Mudd College

Robert Harper, Peter Lee
Carnegie Mellon

ABSTRACT

The goal of the TIL project was to explore the use of Typed Intermediate Languages to produce high-performance native code from Standard ML (SML). We believed that existing SML compilers were doing a good job of conventional functional language optimizations, as one might find in a LISP compiler, but that inadequate use was made of the rich type information present in the source language. Our goal was to show that we could get much greater performance by propagating type information through to the back end of the compiler, without sacrificing the advantages afforded by loop-oriented and other optimizations.

We also confirmed that using typed intermediate languages dramatically improved the reliability and maintainability of the compiler itself. In particular, we were able to use the type system to express critical invariants, and enforce those invariants through type checking. In this respect, TIL introduced and popularized the notion of a certifying compiler, which attaches a checkable certificate of safety to its generated code. In turn, this led directly to the idea of certified object code, inspiring the development of Proof-Carrying Code and Typed Assembly Language as certified object code formats.

1. USING TYPES IN COMPILATION

The Fox Project [3] was started by researchers at Carnegie Mellon to investigate the application of advanced programming languages in critical systems software. We felt that *H*igher-*O*rder, strongly *T*yped (HOT) languages, such as Standard ML [10], could provide the reliability, security, and maintainability desperately lacking in existing systems. However, the performance of code generated by existing SML compilers was not sufficient to meet our needs.

The smooth integration of type inference, polymorphism, algebraic datatypes, and modules, as well as its formal foundations is what set SML apart from other languages. But ironically, a traditional ML compiler (circa 1990) such as as Standard ML of New Jersey (SML/NJ) [1], failed to exploit this sophisticated type system. Once a program had been checked for type safety, the types were simply thrown away, and code was mapped down to a uni-typed lambda-calculus representation, much as one would use for a Scheme or LISP compiler (minus the run-time type tests.) This just seemed wrong!

A key reason why types went unutilized was type abstraction. Consider a polymorphic function to sort arrays

```
val sort: ('a * 'a -> bool) -> 'a array -> 'a array
```

which abstracts the type (`'a`) of the elements of the array. How would a compiler generate code for `sort` when there is no one size, alignment, or calling convention for `'a` values?

An easy solution, and the solution used by most ML compilers of the time, is to force all values into a uniform representation: a machine word. If an object is too large to fit into a word, then it is "boxed" by placing it in memory and using a pointer in its place. Unfortunately, the overheads of using a boxed representation are considerable.

Previous research, notably the coercion work of Leroy [8] which was also implemented in SML/NJ [23], had tried to avoid these overheads. However, Leroy's technique did not apply to recursive or imperative data structures (i.e., lists, trees, and arrays). In a paper presented the previous year, Harper and Morrisett suggested a new approach to compiling polymorphic languages based on *intensional polymorphism* [6]. The idea was to to compile monomorphic code the same way you would in Pascal. For polymorphic code, you would pass in a run-time representation of the unknown types. The code could then look at this run-time type information to determine the information it needed, such as size, alignment, or calling convention. Intensional polymorphism also made it possible to support tag-free, accurate garbage collection and polymorphic equality. Thus, intensional polymorphism seemed to be a promising approach for avoiding boxes and tags.

A large part of the focus of TIL was providing support for intensional polymorphism. To do so required that type information be propagated to the back end of the compiler so that types are available for analysis by polymorphic code. This required the development of type-preserving compilation techniques that translate both the code and its type in such a way that the typing relation is preserved. Work in this direction was initiated by Harper and Lillibridge [4, 5], and continued with the development of typed closure conversion [11, 16], and type-based approaches to garbage collection [14, 15]. Much of Morrisett's dissertation [13] is devoted to developing these ideas.

Apart from supporting intensional type analysis, type-preserving translations make it possible for the compiler to check its own integrity by type checking the results of each major compilation phase. This greatly facilitates team development and long-term maintenance of the compiler by

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

catching many errors at a very early stage. Moreover, these ideas led naturally to the idea of a *certifying compiler*, which augments its object code with a formal representation of its type safety. This innovation is perhaps the most significant lasting contribution of the TIL compiler.

2. INTERACTION WITH OPTIMIZATION

Of course, using type-based methods was not without its risks. While intensional polymorphism could potentially eliminate the overheads of boxing and tagging from monomorphic code, it would also slow down polymorphic code. Furthermore, without a full suite of conventional functional language optimizations, it would be difficult to show convincing speedups even for the monomorphic code. Finally, the requirement to maintain accurate type information throughout compilation raised the question of whether doing so would interfere with the optimizations needed to achieve good performance.

Fortunately, we were able to implement a comprehensive suite of optimizations in a type-preserving fashion. Indeed, TIL went well beyond the state of the art for functional compilers. Function arguments were unboxed; loop-invariant computations were hoisted; array bounds checks were eliminated; closures were avoided through inlining and uncurrying; and of course, polymorphic code was specialized at call-sites to avoid the overheads of run-time type dispatch.

The optimizations were extremely effective, at least for the small benchmarks we used in evaluation. Indeed, they were perhaps *too* effective in the sense that they naturally eliminated *all* of the polymorphism in the programs, through aggressive inlining and specialization. We should have used larger, more realistic benchmarks but we had not yet implemented the SML module system, and most large programs made extensive use of modules. Furthermore, we performed whole program optimization, which does not scale well to large programs. And perhaps the biggest problem was that our compiler took an inordinate amount of time to optimize the code. In the end, there wasn't clear evidence that intensional polymorphism was what lead to the performance improvements compared to this whole-program optimizer [24, 9]

Nonetheless, the TIL compiler showed that, if we were willing to be aggressive in the elimination of polymorphism and other optimizations, we could achieve spectacular speedups, while maintaining accurate type information through to the back end.

3. THE INFLUENCE OF TIL

Later SML compiler efforts, including Flint [22], MLton [12], TILT [21], and Church [2] took the TIL results much further. For example, Flint used a clever combination of Leroy-style coercions and intensional polymorphism to get the best of each approach, and to scale the ideas to the full SML language. The MLton compiler simply eliminated all polymorphism through a whole-program transformation, resulting in a very straightforward, but high-performance compiler. The Church project used an alternative to intensional polymorphism (namely finite intersection and union types) and focused on flow analyses. And the TILT project has focused on a clean, type-preserving methodology for the SML module system.

At the same time TIL was being developed, Java was starting to gain momentum, and we believe it is fair to say that the TIL had some influence on the various proposals for adding generics to Java (c.f. [18, 20]), as well as the current implementation of C# generics [7].

Though performance was the initial focus, it soon became clear that the lasting idea was that we should be using strongly-typed intermediate languages in our compilers. When developing the compiler, we quickly found that the ability to type-check the intermediate code after a transformation was extremely effective for finding bugs. That is, more often than not, an incorrect optimization would manifest itself as typing error in the output.

Making the connection with security and assurance as suggested by the Java Virtual Machine, it soon became clear that if we could type-check the *output* of the compiler, as well as the intermediate stages, then an untrusted consumer could verify key safety properties of native machine code. Thus, TIL led directly to the development of Proof-Carrying Code [19] and Typed Assembly Language [17].

REFERENCES

[1] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science*.

[2] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed reprsentation transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 11–24, Amsterdam, June 1997.

[3] http://www.cs.cmu.edu/f̃ox.

[4] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, SC, January 1993. ACM, ACM.

[5] R. Harper and M. Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(4):361–380, November 1993.

[6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[7] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Snowbird, Utah, June 2001.

[8] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, Jan. 1992.

[9] X. Leroy. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

[10] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[11] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[12] http://www.mlton.org.

[13] G. Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[14] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Functional Programming and Computer Architecture*, pages 66–77, La Jolla, CA, June 1995. ACM.

[15] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. In A. Gordon and A. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press, 1997.

[16] G. Morrisett and R. Harper. Typed closure conversion for recursively-defined functions (extended abstract). In A. Gordon, A. Pitts, and C. Talcott, editors, *Second Workshop on Higher-Order Techniques in Operational Semantics (HOOTS-II)*, volume 10 of *Electronic Notes in Theoretical Computer Science*, Stanford University, Stanford, California, December 1998. Elsevier. URL: http://www.elsevier.nl/locate/entcs/volume10.html.

[17] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–568, May 1999.

[18] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for java. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, 1997.

[19] G. Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[20] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 1997.

[21] L. Peterson, P. Cheng, R. Harper, and C. Stone. Implementing the tilt internal language. Technical Report CMU-CS-00-180, Carnegie Mellon School of Computer Science, Dec. 2001.

[22] Z. Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

[23] Z. Shao and A. Appel. A type-based compiler for Standard ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, June 1995.

[24] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, 1996.