Certifying compilation for Standard ML in a type analysis framework. (Thesis Summary)

Leaf Petersen

April 1, 2004

1 Types in compilation

It is an unfortunate fact about the state of programming that even good programs sometimes go wrong. It is not uncommon for programs to crash or misbehave, whether accidentally, or with malicious intent. This problem has been greatly compounded in recent years by the proliferation of *mobile* code. More and more of the code that is run is downloaded in bits and pieces from various sources. Examples of this include Java-script and Java(TM) programs downloaded into a web browser, applications downloaded directly over the internet, and code run on behalf of others (such as the SETI@HOME project, which uses donated spare processor cycles to further the search for extra-terrestrial life). The proliferation of mobile code is expected only to increase as networked technology becomes more a part of everyday life.

Unfortunately, it is particularly hard to trust the behavior of this sort of code. The code producer may be unknown to the code consumer, or the identity of the producer may be spoofed. Moreover, even trusted producers occasionally produce programs that go wrong.

It would certainly seem desirable to be able to rule out programs that are unsafe. Unfortunately, determining the safety of arbitrary machine code is undecidable. A compromise solution that has been in existence for several decades, is to restrict ourselves to programming in a language which has the property that all programs are safe. The resulting compiled code is therefore safe by construction, *modulo the correctness of the compiler*. Languages such as LISP, ML and Java all have this property: that all programs written in these languages are guaranteed with some level of certainty not to crash.

Unfortunately for the purposes of mobile code, the safety properties enjoyed by these languages are only guaranteed at the source level: once the source code has been compiled to machine instructions, the safety guarantee lies only in the implicit property of being in the image of the safe language under compilation. One solution to this is to ship around instead something tantamount to source code, allowing the consumer to validate the code independently. This is in essence the Java(TM) byte-code solution. This places much of the burden of compilation on the code consumer, who must still in turn trust that their own compiler is correct.

The two most commonly used solutions then, are either to accept arbitrary machine code based on trust in the producer of the code, or to accept only annotated high-level code and to trust the local compiler. Both of these solutions leave much to be desired.

To better this, several systems have been proposed for annotating machine code in such a way that correctness remains *checkable*. Along with the code, the code consumer receives a certificate that can be used to *check* that that the code conforms to the correctness assertions that it claims. The only software that the code consumer must still trust is the checker itself. Two notable examples of this include Proof Carrying Code (PCC) from CMU [Nec98] and Typed Assembly Language (TAL) from Cornell [MWCG98]. The PCC system provides for machine code to be annotated with proofs of safety properties done in first-order logic. The code consumer simply checks that the proofs are indeed correct – a relatively simple procedure. This system is very general, in that it can be used to certify any property which can be expressed in the logic. TAL on the other hand specializes to the particular property of type safety. TAL provides for a machine code which is annotated with types, which can then be type checked by the code consumer.

A certifying compiler is one which produces, along with its normal output, a *certificate* which can be used to check that the generated code is safe according to some policy. Certifying compilers have been written translating safe subsets of C to both PCC and TAL [MCG+99, NL98]. More ambitiously, a full scale Java(TM) compiler has been written targeting PCC [CLN+00].

The **TILT** (TIL Two) compiler is an optimizing compiler developed at CMU that implements the full Standard ML '98 definition and includes support for separate compilation. Important ideas pioneered in **TILT** and its predecessor TIL include using intensional polymorphism [HM95] to reduce the cost of implementing polymorphism and garbage collection. Compilation proceeds as a series of typed transformations into successively lower level typed languages. Type information is used to allow for optimized data representations and to do "almost tag-free" garbage collection. Currently however, type information is mostly erased in **TILT** well before the transformation to machine code is made, and hence safety properties of the resulting code can only be asserted - not checked.

TILT uses types during compilation for optimization purposes, and consequently requires an intermediate language with a very expressive type theory. Previous work on typed assembly languages has primarily focused on preserving type information for certification purposes. I claim that these two uses of low level typed languages are compatible. *A compiler can use types to generate certified binaries while retaining the ability to perform complicated type based optimizations on a full modern language.*

I have demonstrated this by extending the TILT-ML compiler to maintain type information used in the intermediate passes of the compiler all the way through code generation, producing certified binaries without sacrificing the ability to perform type analysis optimizations. This dissertation gives a careful theoretical description of the key elements of the compilation process, and proves soundness theorems for the translations between the major intermediate languages. A description is also given of the actual implementation, including some empirical results. In the next two sections, I provide a more detailed overview of the previously existing (non-certifying) **TILT** infrastructure and give a high-level overview of the certifying **TILT** compiler which is the subject of this dissertation.

2 The non-certifying TILT compiler

The past years have seen a great deal of interest in the idea of "typed compilers" that maintain type information deep into the compilation process. Such type information can be exploited by the compiler internally to allow for optimized data representations and to do tag-free garbage collection, as well as providing the compiler with a basis for internal correctness checks. This work

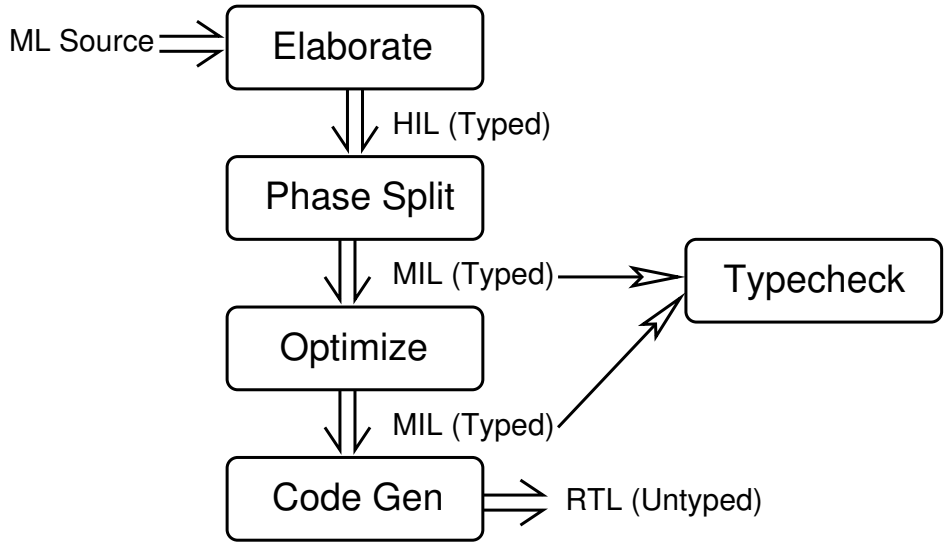

Figure 1: TILT Architecture

was pioneered in the TIL compiler at CMU [TMC+96]. Other recent work has also suggested the possibility of maintaining type information through to the machine code as a form of certification [MWCG97].

The TIL compiler clearly demonstrated that typed compilation was both feasible and desirable. However, TIL compiled only the core language of Standard ML: the powerful modular features that are one of the most important elements of SML were not dealt with. The TIL Two (**TILT**) compiler was aimed at addressing this shortcoming.

Figure 1 depicts the structure of the non-certifying **TILT** compiler. It's architecture is based around two typed intermediate languages. The initial elaboration from SML source targets a structures calculus called the HIL (High Intermediate Language). This language is relatively close to SML, and among other things provides the interface language used for separate compilation. After elaboration (and hence typechecking), the HIL is translated to a second typed language called the **MIL** (Middle Intermediate Language) through a process called phase splitting [HMM90]. The phase splitting process maps each SML structure into separate type and term level records, representing the static and dynamic portions of the structure. Similarly, SML functors are mapped to type and term level functions. In this fashion, modular programs are translated into programs containing only lambda calculus terms.

The **MIL** is the language in which almost all of the optimization passes implemented in **TILT** are done. This constrains the design of the **MIL**, since it must be possible to express the results all of the desired optimizations in a typed fashion. In particular, it is important that primitives for data representation optimizations be present at this level. By "hiding" type analysis inside of a few primitives, the **MIL** avoids the need for a general typecase construct as used in the λ_i^{ML} calculus. Nonetheless, the fact that some **MIL** primitives do indeed analyze their types mandates a type passing interpretation for the **MIL** operational semantics.

All of the intermediate languages of the **TILT** compiler up to and including the **MIL** are typed, and all of the compiler passes on these languages are type-preserving in the sense that they map well-typed programs to well-typed programs. Unfortunately for certification purposes,

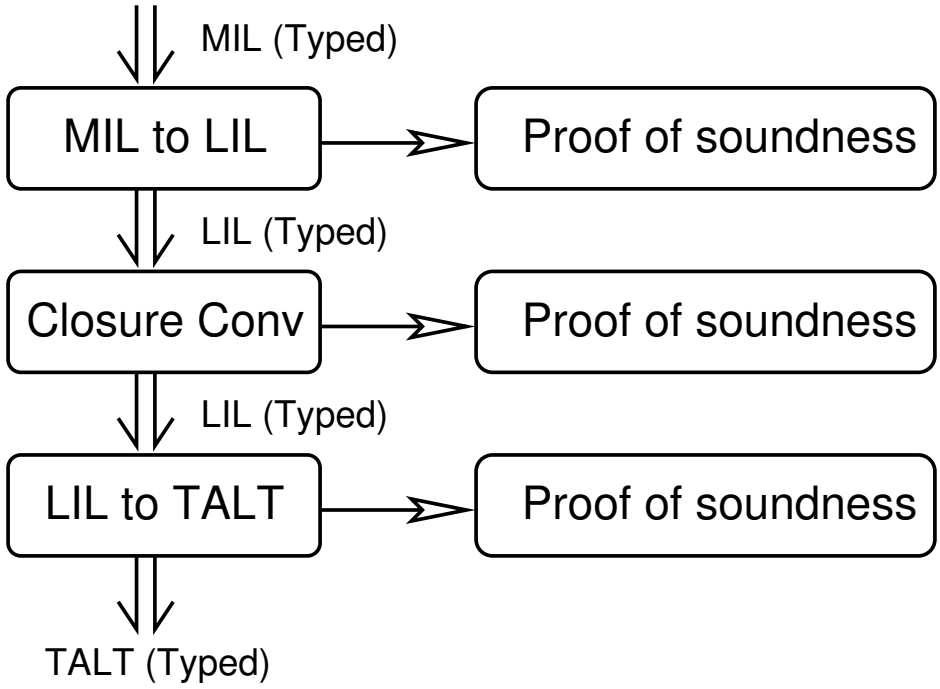

Figure 2: Structure of the theoretical compiler

the subsequent languages from the **MIL** on down are not typed, and hence the generated code cannot in general be proven safe. This dissertation replaces this un-typed backend with a new type-preserving backend that produces certified code.

3 The certifying TILT compiler

One of the major goals of certifying compilation is to ensure that the certifying compiler is type-preserving: that is, that it maps well-typed programs in the source language to well-typed programs in the target language. In order to show that this is the case, I spend the first part of this dissertation presenting idealized versions of the compiler intermediate languages, and proving the soundness of the translations between them. The next two sections describe the idealized compiler and it's relation to the implementation.

3.1 The theoretical compiler

The theoretical portion of this dissertation describes the framework for a translation mapping the original **TILT** internal language (the **MIL**, described in chapter 2) down to a typed assembly language that I call **TALT** (Typed Assembly Language for **TILT**). This translation uses as an intermediate stage a new internal language called the **LIL** (Low Level Language). The **LIL** is an impredicatively typed lambda calculus based on Crary and Weirich's LX [CW99]. Figure 2 describes the structure of the theoretical compiler.

The **LIL**, described in chapter 3, provides a very rich type system in which the type analysis of **TILT** can be represented using term level constructs. In addition to various engineering benefits,

this fact allows us to take a type erasure interpretation, instead of the type passing interpretation apparently mandated by the **MIL** primitives. The fact that we can embed type analysis into the term level reflects in a typed fashion exactly the techniques already used in an untyped fashion for implementing type passing languages. Chapter 4 gives a brief introduction to the type analysis methodology of the **LIL** via a worked example.

The translation from **MIL** to **LIL** serves to make type analysis and type representations explicit in the term language. This translation is described in detail in chapter 5. A subsequent pass mapping **LIL** terms to **LIL** terms performs closure conversion. The closure conversion translation is described in chapter 6.

Finally, I give a translation from the **LIL** into an idealized typed assembly language called **TALT**. **TALT** code, is essentially machine code with type annotations added allowing it to be typechecked. Currently, there are in addition to the standard assembly language instructions, several typed primitives corresponding to assembler macros. These primitives handle memory allocation (and hence the interaction with the garbage collector) and array bounds checking. The **TALT** language is introduced in chapter 7, and the translation from **LIL** programs to **TALT** programs is given in chapter 8.

The theoretical **LIL** is actually very close to the **LIL** as implemented in the compiler, mostly lacking only an extended set of basic primitive operations. The presentation of the **TALT** target language is intended to suggest how the ideas used to implement type analysis in the **LIL** translate down to the assembly code level. The actual **TALx86** implementation departs significantly from that given here. However, for the most part the theoretical translations between the various languages are designed to capture all or most of the essential details of the implementation. In particular, the **LIL** to **TALT** translation of chapter 8 attempts as much as possible to account for the actual code generation techniques used by the implementation.

The main result of the theoretical portion of this dissertation is a proof that the compilation of **LIL** programs into **TALT** programs preserves types, in the sense that each of the individual translations is proved to map well-typed terms to well-typed terms. Proofs of soundness for the various translation phases are given in the chapters in which they are defined.

3.2 Compiler implementation

The second half of this thesis is an actual implementation of a certifying compiler for Standard ML. The theoretical compiler discussed in the previous section is intended as a model for the implementation. Where the original architecture (see figure 1) switches to an untyped language, the new implementation must instead take the typed output and continue the compilation process in a typed setting, as shown in figure 3. The certifying **TILT** implementation is discussed in chapter 9.

The implementation of certifying **TILT** required the addition of essentially five additional compiler stages.

- **MIL** singleton elimination
- Translation to **LIL**
- Closure conversion
- Optimization

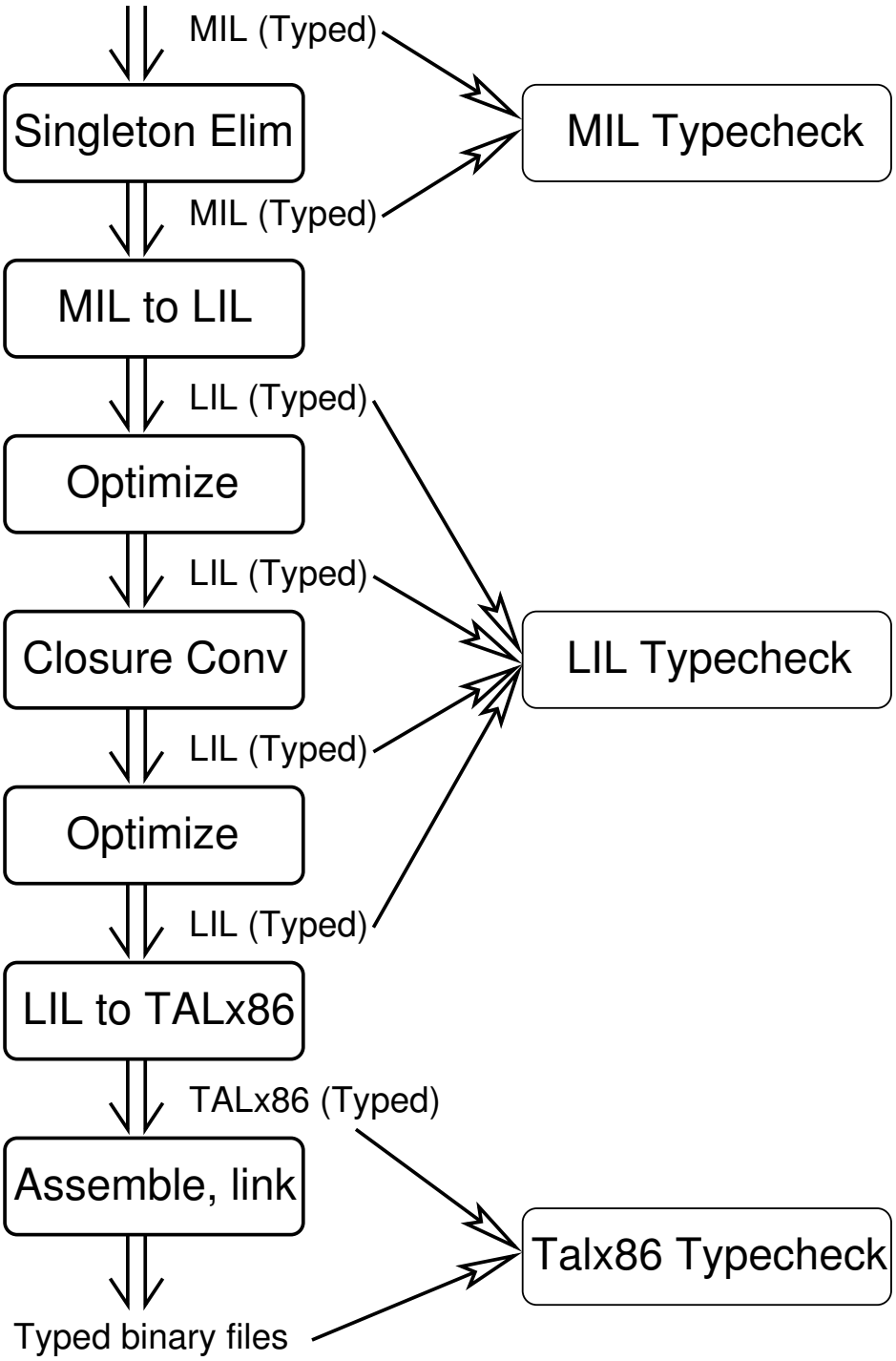

Figure 3: Structure of the certifying **TILT** backend

- Translation to **TALx86**

To simplify the meta-theory of the **LIL**, I have chosen not to support the singleton kinds of the original **MIL** as implemented in the compiler. Therefore, before (or concurrent with) the translation to **LIL**, singletons must be eliminated from **MIL** code. A proof that this is possible, and a simple algorithm for doing so has been given by Crary [Cra00], but some work may be necessary to make a practical version of this that does not increase code size unnecessarily.

The translation to **LIL** is described in detail in chapter 5. The most interesting part of this translation is the process of making type analysis explicit. The **MIL** has no general notion of type analysis, instead using type analyzing primitives to implement specific optimizations such as floating point array unboxing and flattening function arguments into records. The translation to **LIL** compiles these primitives into uses of a general type analysis mechanism.

The **MIL** implements all of the optimizations that **TILT** supports, so extended optimization of **LIL** code is mostly unnecessary. However, the translations tend to produce code that can be improved significantly by a simple optimizer: for example, the closure converter frequently introduces dead code and projections from known records. Rather than making other phases do significantly more work to avoid this, it was simpler and cleaner to implement a basic optimization pass for the **LIL**. Note that the optimizer does not have a counterpart in the theoretical model.

It would be appealing to rely as well on the **MIL** closure converter, and only translate closure converted code. Unfortunately, the **MIL** notion of closure conversion is not easily compatible with the **LIL** notion, since the **MIL** must closure-convert types as data. Translating closure converted **MIL** code would require "de-closure converting" types, which is not appealing. For this reason, a separate closure conversion pass for the **LIL** was implemented, closely modeled on the formal closure conversion translation from chapter 6.

In order to allow intermediate code to be validated, it was also useful to implement a type checker for the **LIL** language. This allows the output of the various phases of the compiler to be checked for errors internally. The ability to check type correctness of internal program representations has proved valuable in the development of **TILT**.

Lastly, a translation is given mapping **LIL** programs into **TALx86** programs. Note that here, the implementation diverges significantly from the formal model described in chapter 8, in that the **TALT** and **TALx86** languages are quite different. However, the theoretical model was carefully designed to capture many of the interesting algorithmic approaches used in the actual implementation.

The final portion of the dissertation gives a basic quantitative evaluation of the implementation, including measurements of the size of the generated code and certificates to get a feeling for the overhead of certification. Some measurements of run time behavior of the compiled programs are also given.

It is important to state here that while I measure these quantities to gain some understanding of the cost of my approach, optimizing for small certificates and fast certification time is beyond the scope of this dissertation. The purpose of this work is to demonstrate the feasibility of using types to certify type analyzing code generated from a full-scale, general purpose language. Engineering the representations, while certainly important for making the compiler practical, was not a primary goal.

References

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 95–107, Vancouver, Canada, June 2000. ACM Press.

[Cra00] Karl Crary. Sound and complete elimination of singleton kinds. Technical Report CMU-CS-00-104, School of Computer Science, Carnegie Mellon University, 2000.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, Paris, France, September 1999. ACM Press.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[MWCG97] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. Technical Report TR97-1651, Department of Computer Science, Cornell University, 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.