Scalable Certification for Typed Assembly Language

Dan Grossman and Greg Morrisett *

Department of Computer Science, Cornell University

Abstract. A type-based certifying compiler maps source code to machine code and target-level type annotations. The target-level annotations make it possible to prove easily that the machine code is type-safe, independent of the source code or compiler. To be useful across a range of source languages and compilers, the target-language type system should provide powerful type constructors for encoding higher-level invariants. Unfortunately, it is difficult to engineer such type systems so that annotation sizes are small and verification times are fast.

In this paper, we describe our experience writing a certifying compiler that targets Typed Assembly Language (TALx86) and discuss some general techniques we have used to keep annotation sizes small and verification times fast. We quantify the effectiveness of these techniques by measuring their effects on a sizeable application — the certifying compiler itself. Using these techniques, which include common-subexpression elimination of types, higher-order type abbreviations, and selective reverification, can dramatically change certificate size and verification time.

1 Background

A certifying compiler takes high-level source code and produces target code with a *certificate* that ensures that the target code respects a desired safety or security policy. To date, certifying compilers have primarily concentrated on producing certificates of type safety. For example, Sun's `javac` compiler maps Java source code to statically typed Java Virtual Machine Language (JVML) code. The JVML code includes type annotations that a verifier based on dataflow analysis can use to ensure that the code is type-safe.

However, both the instructions and the type system of JVML are at a relatively high level and are specifically tailored to Java. Consequently, JVML is ill-suited for compiling a variety of source-level programming languages to high-performance code. For example, JVML provides only high-level method-call and

* This material is based on work supported in part by the AFOSR grant F49620-97-1-0013, ARPA/RADC grant F30602-1-0317, and a National Science Foundation Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

method-return operations. Also, it provides no provision for performing general tail-calls on methods. Therefore, JVML is a difficult target for compilers of functional languages such as Scheme that require tail-call elimination.

In addition, current platforms for JVML either interpret programs or compile them further to native code. Achieving acceptable performance seems to demand compilation with a good deal of optimization. To avoid security or safety holes, the translation from JVML to native code should also be done by a certifying compiler. That way, we can verify the safety of the resulting code instead of trusting the "just-in-time" compiler.

Another example of a certifying compiler is Necula and Lee's Touchstone compiler [22]. Touchstone compiles a small, type-safe subset of C to optimized DEC-Alpha assembly language. The key novelty of Touchstone is that the certificate it produces is a formal "proof" that the code is type-correct. Checking the proof for type-correctness is relatively easy, especially compared to the *ad hoc* verification process for JVML. As such, the Touchstone certificates provide a higher degree of trustworthiness.

The proofs of the Touchstone system are represented using the general-purpose logical framework LF [11]. The advantage of using LF to encode the proofs is that, from an implementation perspective, it is easy to change the type system of the target language. In particular, the proof checker is parameterized by a set of primitive axioms and inference rules that effectively define the type system. The checker itself does not need to change if these rules are changed. Consequently, the use of LF makes it easy to change type systems to adapt to different source languages or different compilation strategies. Indeed, more recent work uses a very different type system for certifying the output of Special J [5, 24], a compiler for Java.

Although changing the type system is easy for the implementor, doing so obligates one to an enormous proof burden: Every change requires a proof of the soundness of the type system with respect to the underlying machine's semantics. Constructing such proofs is an extremely difficult task. In the absence of a proof, it is not clear what assurances a verifier is actually providing.

1.1 An Alternative Approach

Our goal is to make it easy for certifying compilers to produce provably type-correct code without having to change the type system of the target language. That way, it suffices to write and trust one verifier for one type system. Toward this end, we have been studying the design and implementation of general-purpose type systems suitable for assembly language [20, 19]. Ultimately, we hope to discover typing constructs that support certifying compilation of many orthogonal programming-language features.

Our current work focuses on the design of an extremely expressive type system for Intel's IA32 assembly language and a verifier we call TALx86 [18]. Where possible, we have avoided including high-level language abstractions like procedures, exception handlers, or objects. In fact, the only high-level operation that is a TALx86 primitive is memory allocation. We also have not "baked in" compiler-specific abstractions such as activation records or calling conventions. Rather, the type system of TALx86 provides a number of primitive type constructors, such as parametric polymorphism, label types, existential types, products, recursive types, etc., that we can use to encode language features and compiler invariants. These type constructors have either been well studied in other contexts or modeled and proven sound by our group.

In addition, we and others have shown how to encode a number of important language and compiler features using our type constructors. For example, our encoding of procedures easily supports tail-call optimizations because the control-flow transfers are achieved through simple machine-level jumps. In other words, we do not have to change the type system of TALx86 to support these optimizations. Type soundness of TALx86 ensures that compilers targeting TALx86 produce only code with safe run-time behavior. Some specific assurances are that the program counter will always point to executable code, unallocated memory will never be dereferenced, and system routines (such as input/output) will never be called with inappropriate arguments. In these respects, TALx86 provides an attractive target for certifying compilers.

1.2 The Problem

Unfortunately, there is a particularly difficult engineering tradeoff that arises when a certifying compiler targets a general-purpose type system like TALx86: Encoding high-level language features, compiler invariants, and optimizations into primitive type constructors results in extremely large types and type annotations — *often much larger than the code itself.* Thus, there is a very real danger that our goal of using one general-purpose type system will be defeated by practical considerations of space and time.

The work presented here is a case study in writing a certifying compiler that targets the general-purpose typed assembly language TALx86. The source language for our compiler, called Popcorn, shares much of its syntax with C, but it has a number of advanced language features including first-class parametric polymorphism, non-regular algebraic datatypes with limited pattern matching, function pointers, exceptions, first-class abstract data types, modules, etc. Indeed, the language is suitably high-level that we have ported various ML libraries to Popcorn without needing to change their structure substantially. The certifying compiler for Popcorn is itself written in Popcorn.

Although the TALx86 type system is very expressive, it is far from being able to accept all safe assembly programs. However, we have found that it is expressive enough to allow a reasonable translation of Popcorn's linguistic features. Because the compiler's invariants are encoded in the primitive typing constructs of TALx86, the most difficult aspect of efficient, scalable verification is handling the potentially enormous size of the target-level types. We use our experience to suggest general techniques for controlling this overhead that we believe transcend the specifics of our system. The efficacy of these techniques is demonstrated quantitatively for the libraries and compiler itself. In particular, the size of the type annotations and the time needed to verify the code are essentially linear in the size of the object code. The constant factors are small enough to permit verification of our entire compiler in much less than one minute.

In the next section, we give a taxonomy for general approaches to reducing type-annotation overhead and further discuss other projects related to certifying compilation. Although it is an informal description of existing techniques, we have found this classification useful and we know of no other attempt to classify the approaches.

In Section 3, we summarize relevant aspects of the TALx86 type system, annotations, and verification process. We then show how these features are used to encode the provably safe compilation of the control-flow aspects of Popcorn, including procedures and exceptions. This extended example demonstrates that an expressive type system can permit reasonable compilation of a language for which it is not specifically designed. It also shows qualitatively that if handled naively, type-annotation size becomes unwieldy.

In Section 4, we use the example to analyze several approaches that we have examined for reducing type-annotation overhead. Section 5 presents the quantitative results of our investigation; we conclude that the TALx86 approach scales to verify our Popcorn compiler, the largest Popcorn application we have written. Moreover, all of the techniques contribute significantly to reducing the overhead of certifying compilation. Finally, we summarize our conclusions as a collection of guidelines for designers of low-level safety policies.

2 Approaches to Efficient Certification

Keeping annotation size small and verification time fast in the presence of optimizations and advanced source languages is an important requirement for a practical system that relies on certified code. In this section, we classify some approaches to managing the overhead of certifying compilation and discuss their relative merits. None of the approaches are mutually exclusive; any system will probably have elements of all of them.

The "Bake it in" Approach If the type system supports only one way of compiling something, then compilers do not need to write down that they are using that way. For example, the type system could fix a calling convention and require compilers to group code blocks into procedures. JVML, Touchstone, and Special J all use this approach.

Baking in assumptions about procedures eliminates the need for any annotations describing the interactions between procedures. However, it inhibits some inter-procedural optimizations, such as inter-procedural register allocation, and makes it difficult to compile languages with other control features, such as exception handlers. In general, the "bake it in" approach reflects particular source features into the target language rather than providing low-level constructors suitable for encoding a variety of source constructs. For example, the certifier for Special J first processes a, "class descriptor whose form is very close to that of the JVM class descriptors" [24], so only programs conforming to the JVML class hierarchy and type system are certifiable by this checker.

Even general frameworks inevitably bake in more than the underlying machine requires. A TALx86 example is that labels are abstract — well-formed code cannot examine the actual address used to implement a label. This abstraction prevents some clever implementation techniques. Any verifiable safety policy must impose some conservative restrictions; choosing the restrictions is a crucial design decision that is a fundamental part of a policy.

The "Don't optimize" Approach If a complicated analysis is necessary to prove an optimization safe, then the reasoning involved must be encoded in the annotations. For example, when compiling dynamically typed languages such as Scheme, dynamic type tests are in general necessary to ensure type safety. A simple strategy is to perform the appropriate type test before every operation. With this approach, a verifier can easily ensure safety with a minimum of annotations. This strategy is the essence of the verification approach suggested by Kozen [14]. Indeed, it results in relatively small annotations and fast verification, but at the price of performance and flexibility.

In contrast, an optimizing compiler may attempt to eliminate the dynamic checks by performing a "soft-typing" analysis [30]. However, the optimized code requires a more sophisticated type system to convince the verifier that type tests are unnecessary. To make verification tractable, such type systems require additional annotations. For example, the Touchstone type system supports static elimination of array-bounds checks, but it requires additional invariants and proof terms to support the optimization.

Another example is record initialization: An easy way to prove that memory is properly initialized is to write to the memory in the same basic block in which the memory is allocated. Proving that other instruction schedules are safe may require dataflow annotations that describe the location of uninitialized memory.

Unoptimized code also tends to be more uniform, which in turn makes the annotations more uniform. For example, if a callee-save register is always pushed onto the stack by the callee (even when the register is not used), then the annotations that describe the stack throughout the program will have more in common. Such techniques can improve the results of the "Compression" approach (discussed below) at the expense of efficiency.

The "Reconstruction" Approach If it is easy for the verifier to infer a correct annotation, then such annotations can be elided. For example, Necula shows how simple techniques may be used for automatically reconstructing large portions of the proofs produced by the Touchstone compiler [23].

It is important that verification time not unduly suffer, however. For this reason, code producers should know the effects that annotation elision can have. Unfortunately, in expressive systems such as TALx86, many forms of type reconstruction are intractable or undecidable. The verifier could provide some simple heuristics or default guesses, but such maneuvers are weaker forms of the "bake it in" approach.

A more extreme approach to reconstruction would be to include a general-purpose theorem prover in the verification system. Unless the prover generates proofs that are independently checked, the trusted computing base would become larger and more complex. Any generated proofs would need to be concise as well. The TALx86 project has maintained the design goal that type-checking should be essentially syntax-directed; search and backtracking seem beyond the realm of efficient verification. However, recent work by Necula and Rahul [24] suggests using annotations not to provide a proof, but instead to guide the prover's non-determinism. In essence, the insight is that a compiler that knows enough about the verifier's decision procedure can guide reconstruction to avoid the overhead of search.

Certification systems invariably use reconstruction when the type of a construct is straightforward to compute from the types of its parts. For example, explicitly typed source languages never require explicit types for every term; these types are reconstructed from the explicit types of variables. Similarly, low-level systems do not explicitly describe how every single instruction changes the abstract state of the program. For most instructions, it is just as efficient to examine the instruction and recompute this information.

The "Compression" Approach Given a collection of annotations, we could create a more concise representation that contains the same information. One technique for producing a compact wire format is to run a standard program such as `gzip` on a serialized version. If the repetition in the annotations manifests itself as repetition in the byte stream, this technique can be amazingly effective (see Section 5). However, it does not help improve the time or space required for verification if the byte stream is uncompressed prior to processing.

A slightly more domain-specific technique is to create a binary encoding that shares common subterms between annotations. This approach is effectively common-subexpression elimination on types. Because the verifier is aware of this sharing, it can exploit it to consume less space. There is an interesting tradeoff with respect to in-place modification, however. If a simplification (such as converting an annotation to a canonical form for internal use) is sound in all contexts, then it can be performed once on the shared term. However, if a transformation is context-dependent, the verifier must make a copy in the presence of sharing.

Work on reducing the size of JVML annotations has largely followed the compression approach [25, 2]. For example, projects have found ways to exploit similarities across an entire archive of class files. Also, they carefully design the wire format so that downloading and verification may be pipelined. The TALx86 encoding does not currently have this property, but there is nothing essential to the language that prevents it.

Shao and associates [26] have investigated the engineering tradeoffs of sharing in the context of typed intermediate languages. They suggest a consistent use of hash-consing (essentially on-line common-subexpression elimination) and suspension-based lambda encoding [21] as a solution. Their hash-consing scheme also memoizes the results of type reductions so that identical reductions in the future require only retrieving the answer from a table. The problem of managing low-level types during compilation is quite similar to the problem of managing them during verification, but in the case of type-directed compilation, it is appropriate to specialize the task to the compiler.

Finally, we should note that comparing the size of compressed low-level types to the size of uncompressed object code is somewhat misleading because object code compresses quite well [16, 7]. Domain-specific techniques include taking the instruction format into account (instead of the generic compression technique of processing entire bytes); detecting common sequences of instructions; and detecting similarity modulo a rarely repeated field, such as a branch target address. Analogous techniques may prove useful for annotations as well, but we know of no work that has tried them.

The "Abbreviation" Approach The next step beyond simple sharing is to use *higher-order* annotations to factor out common portions. Such annotations are essentially functions at the level of types. Tarditi and others used this approach in their TIL compiler [28]. As we show in Section 4, this approach can exploit similarities that sharing cannot. Furthermore, higher-order annotations make it relatively easy for a compiler writer to express high-level abstractions within the type system of the target language. In our experience, using abbreviations places no additional burden on the compiler writer because she is already reasoning in terms of these abstractions. However, if the verifier must expand the abbreviations in order to verify the code, verification time may suffer.

Higher-order abbreviations are also an important component in the certified-code framework that Appel and Felty [1] propose. They suggest formalizing a machine's semantics and a safety policy in a higher-order logical framework. The code producer must then supply a formal proof that a program obeys the policy. Because a proof expressed directly in terms of a machine's semantics would presumably be enormous, Appel and Felty suggest that a compiler would first prove that a collection of lemmas are sound with respect to the semantics and then apply the lemmas to a program. In a sense, these lemmas are parameterized abbreviations that define a suitably concise type system.

In our system, we use *all* of these approaches to reduce annotation size and verification time. However, we have attempted to minimize the "bake it in" and "don't optimize" approaches in favor of the other techniques. Unlike `javac`, Touchstone, or Special J, TALx86 makes no commitment to calling convention or data representation. In fact, it has no built-in notion of functions; all control flow is just between blocks of code. The design challenge for TALx86, then, is to provide generally useful constructors that compilers can use in novel ways to encode the safety of their compilation strategies.

As a type system, TALx86 does "bake in" more than a primitive logical description of the machine. For example, it builds in a distinction between integers and pointers. As a result, programs cannot use the low bits of pointers to store information and then mask these bits before reading memory. Also, memory locations are statically divided into code and data (although extensions support run-time code generation [12]). In order to investigate the practicality of expressive low-level safety policies, we have relied on a rigorous, hand-written proof of type soundness and a procedural implementation of the verifier. A more formal approach would be to encode the proof in a logical framework and use a verifier produced mechanically from the proof.

Using our approach, we have been able to examine the feasibility of compiler-independent safety policies on a far larger scale than has been previously possible. To date, certifiers based on proof-carrying code technology have all had compiler-specific safety policies and no compiler has targeted the compiler-independent safety policies of Appel and Felty. Not only was TALx86 designed to be compiler-independent, but we and others have written three separate compilers that target TALx86. In this paper, we discuss our optimizing Popcorn compiler.[1] This compiler, itself a certified TALx86 program, is a several-hundred kilobyte executable compiled from over eighteen thousand lines of source code.

3 Compiling to TALx86: An Extended Example

In this section, we briefly review the structure of the TALx86 type system, its annotations, and the process of verification. In what follows, we present relevant TALx86 constructs as necessary, but for the purposes of this paper, it is sufficient to treat the types as low-level syntax for describing pre-conditions. Our purpose is not to dwell on the artifacts of TALx86 or its relative expressiveness. Rather, we want to give some intuition for the following claims, which we believe transcend TALx86:

– If the safety policy does not bake in data and control abstractions, then the annotations that the compiler uses to encode them can become large.
– In fact, the annotations describing compiler conventions consume much more space than the annotations that are specific to a particular source program.
– Although the annotations for compiler conventions are large, they are also very uniform and repetitious, though they become much less so in the presence of optimizations.

Because of this focus, we purposely do not explain some aspects of the annotations other than to mention the general things they are encoding. The reader interested in such details should consult the literature [20, 19, 10, 18, 9, 27].

A TALx86 object file consists of IA32 assembly-language instructions and data. As in a conventional assembly language, the instructions and data are organized into labeled sequences. Unlike conventional assembly language, some labels are equipped with a type annotation. The type annotations on the labels of instruction sequences, called *code types*, specify a pre-condition that must be satisfied before control may be transferred to the label. The pre-condition specifies, among other things, the types of registers and stack slots. For example, if the code type annotating a label `L` is {`eax:int4, ebx:S(3), ecx: ^`

[1] The other compilers are a simple stack-based compiler for Popcorn and a compiler for a core subset of Scheme.

`*[int4,int4]`}, then control may be transferred to the address `L` only when the register `eax` contains a 4-byte integer, the register `ebx` contains the integer value 3, and the register `ecx` contains a pointer (`^`) to a record (`*[...]`) of two 4-byte integers.

Verification of code proceeds by taking each labeled instruction sequence and building a typing context that assumes registers have values with types as specified by the pre-condition. Each instruction is then type-checked, in sequence, under the current set of context assumptions, possibly producing a new context. For most instructions, the verifier automatically infers a suitable typing post-condition in a style similar to dataflow analysis or strongest post-conditions. Some instructions require additional annotations to help the verifier. For example, it is sometimes necessary to explicitly coerce values to a supertype, or to explicitly instantiate polymorphic type variables.

Not all labels require type annotations. However, code blocks without annotations may be checked multiple times under different contexts, depending on the control-flow paths of the program. To ensure termination of verification, the type-checker requires annotations on labels that are moved into a register, the stack,[2] or a data structure (such as a closure); on labels that are the targets of backwards branches (such as loop headers); and on labels that are exported from the object file (such as function entry points). These restrictions are sufficient for verification to terminate. We discuss labels without explicit types in more detail in Section 4.3.

As in a conventional compiler, our certifying compiler translates the high-level control-flow constructs of Popcorn into suitable collections of labeled instruction sequences and control transfers. For present purposes, control flow in Popcorn takes one of three forms:

– an intra-procedural jump
– a function call or return
– an invocation of the current exception handler

Currently, our compiler performs only intra-procedural optimizations, so the code types for function-entry labels are quite uniform and can be derived systematically from the source-level function's type. For simplicity, we discuss these code types first. We then discuss the code types for labels internal to functions, focusing on why they are more complicated than function entries. We emphasize that the distinction between the different flavors of code labels (function entries, internal labels, exception handlers) is a Popcorn convention encoded in the pre-conditions and is in no way specific to TALx86. Indeed, we have constructed other toy compilers that use radically different conventions.

3.1 Function-Entry Labels

As a running example, we consider a Popcorn function `foo` that takes one parameter, an `int`, and returns an `int`. The Popcorn type `int` is compiled to the

[2] Return addresses are an important exception; they do not need explicit types.

TALx86 type `int4`. Arithmetic operations are allowed on values of this type; treating them as pointers is not. Our compiler uses the standard C calling convention for the IA32 architecture. Under this convention, the parameters are passed on the stack, the return address is shallowest on the stack, the return value is passed in register `eax`, and the caller pops the parameters upon return. All of these specifics are encoded in TALx86 by giving `foo` this pre-condition:

```
foo: ∀s:Ts. {esp: { eax: int4
                    esp: int4::s}
                  ::int4
                  ::s}
```

The pre-condition for `foo` concerns only `esp` (the stack pointer) and requires that this register point to a stack that contains a return address (which itself has a pre-condition), then an `int4` (*i.e.* the parameter), and then some stack, `s`. The return address expects an `int4` in register `eax` and the stack to have shape `int4::s`. (The `int4` is there because the caller pops the parameters.) The pre-condition is polymorphic over the "rest" of the stack as indicated by the universal quantification over the stack-type variable `s`. This technique allows a caller to abstract the current type of the stack upon entry, and it ensures that the type is preserved upon return. Types in TALx86 are classified into kinds (types of types), so that we do not confuse "standard" types such as `int4` with "non-standard" types such as stack types. To maintain the distinction, we must label the bound type variable `s` with its kind (`Ts`).

Notice that our annotation already includes much more information than it would need to if the safety policy dictated a calling convention. In that case, we would presumably just give the parameter types and return type of the function. Some systems, including the certifier for Special J [5], go even further — they encode the types in the string for the label, so it appears that no annotation is necessary. Of course, the safety policy now attaches specific meaning to the characters in a label; the annotations are encoded in the assembly listing.

Our annotation does not quite describe the standard C calling convention. In particular, the standard requires registers `ebx`, `esi`, and `edi` to be callee-save. (It also requires `ebp`, traditionally the frame pointer, to be callee-save. Our compiler uses `ebp` for the exception handler.) We encode callee-save registers using polymorphism:[3]

```
foo: ∀s:Ts a1:T4 a2:T4 a3:T4 .
 {esp: {eax:int4 esp: int4::s ebx:a1 esi:a2 edi:a3}
       ::int4::s
  ebx:a1 esi:a2 edi:a3}
```

This pre-condition indicates that for any standard types `a1`, `a2`, `a3`,[4] the appropriate registers must have those types before `foo` is called and again when

[3] Here and below, underlining is only for emphasis.

[4] The kind `T4` includes all types whose values fit in a register.

the return address is invoked. This annotation restricts the behavior of `foo` to preserve these registers because it does not know of any other values with these types. More formally, this fact follows from parametricity [6, 29]. Notice that if we wish to use different conventions about which registers should be callee-save, then we need to change only the pre-condition on `foo`. In particular, we do not need to change the underlying type system of TALx86.

Much more detail is required to encode our compiler's translation of exception handling [19], so we just sketch the main ideas. We reserve register `ebp` to point into the middle of the stack where a pointer to the current exception handler resides. This handler expects an exception packet in register `eax`. Because `foo` might need to raise an exception, its pre-condition must encode this strategy. Also, it must encode that if `foo` returns normally, the exception handler is still in `ebp`. We express all these details below, where `@` is an infix operator for appending two stack types.

```
foo: ∀s1:Ts s2:Ts a1:T4 a2:T4 a3:T4 .
 {esp: {eax:int4
        esp: int4::s1@{esp:s2 eax:exn}::s2
        ebp: {esp:s2 eax:exn}::s2
        ebx:a1 esi:a2 edi:a3}
       ::int4::s1@{esp:s2 eax:exn}::s2}
  ebp: {esp:s2 eax:exn}::s2
  ebx:a1 esi:a2 edi:a3}
```

We urge the reader not to focus on the details other than to notice that none of the additions are particular to `foo`, nor would it be appropriate for a safety policy to bake in this specific treatment of exception handlers. Also, we have assumed there is a type `exn` for exception packets. TALx86 does not provide this type directly, so our compiler must encode its own representation using an extensible sum [9]. Each of the four occurrences of `exn` above should in fact be replaced by the type

```
∃c:Tm ^*[(^T^rw(c)*[int4^rw])^rw,c]
```

but in the interest of type-setting, we spare the reader the result.

For the sake of completeness, we offer a final amendment to make this pre-condition correct. Our compiler schedules function calls while some heap records may be partially initialized. This strategy is arguably better than the "don't optimize" approach of always initializing records within a basic block, but it requires that we convince the verifier that no aliases to partially initialized records escape. In particular, the pre-condition for `foo` uses two *capability variables* [27], as shown below,[5] to indicate that it does not create any aliases to partially initialized records reachable from the caller or exception handler.

[5] The constructor `&[...]` joins two capabilities to produce a harder-to-satisfy capability; we omit its definition.

```
foo: ∀s1:Ts s2:Ts e1:Tcap e2:Tcap a1:T4 a2:T4 a3:T4 .
 {esp: {eax:int4
        esp: int4::s1@{esp:s2 eax:exn cap:e2}::s2
        ebp: {esp:s2 eax:exn cap:e2}::s2
        ebx:a1 esi:a2 edi:a3
        cap: &[e1,e2]}
       ::int4::s1@{esp:s2 eax:exn cap:e2}::s2}
  ebp: {esp:s2 eax:exn cap:e2}::s2
  ebx:a1 esi:a2 edi:a3
  cap: &[e1,e2]}
```

In short, because our compiler has complicated inter-procedural invariants, the naive encoding into TALx86 is anything but concise. (The unconvinced reader is invited to encode a function that takes a function pointer as a parameter.) However, the only parts particular to our example function `foo` are the return type, which is written once, and the parameter types, which are written twice. Even these parts are the same for all functions that take and return integers.

3.2 Internal Labels

In this section, we present the pre-conditions for labels that are targets of intra-procedural jumps. For simplicity, we consider only functions that do not declare any local exception handlers. This special case is by far the most common, so it is worth considering explicitly. Because our compiler does perform intra-procedural optimizations, most relevantly register allocation, the pre-conditions for internal labels are less uniform than those for function-entry labels. Specifically, they must encode several properties about the program point that the label designates:

– A local variable may reside in a register or on the stack.
– Some stack slots may not hold live values, so along different control-flow paths to the label, a stack slot may have values of different types.
– Some callee-save values may reside on the stack while others remain in registers.
– Some heap records may be partially initialized.

First we describe the relevant aspects of our term translation: Any callee-save values that cannot remain in registers are stored on the stack in the function prologue and restored into registers in the function epilogue. The space for this storage is just shallower than the return address. Local variables that do not fit in registers are stored in "spill slots" that are shallowest on the stack. The number of spill slots remains constant in the body of a function. This strategy is fairly normal, but it is far too specific to be dictated by TALx86. Indeed, our original Popcorn compiler did not perform register allocation; it simply pushed and popped variables on the stack as needed.

The pre-condition for internal labels gives the type and location (register or spill slot) for each live local variable. If a stack slot is not live, we must still give it some “place-holder” type so that the stack type describes a stack of the correct size. Different control-flow paths may use the same stack slot for temporary variables of different types. In these cases, no previously seen type can serve as this place-holder. TALx86 provides a primitive type `top4` which is a supertype of all types ranging over word-sized values. We give this type to the dead stack slots at the control-flow join; the appropriate subtyping on control transfers is handled implicitly by the verifier.[6]

In addition to live variables, all of the invariants involving the stack, the exception handler, etc. must be preserved as control flows through labels, so this information looks much as it does for function-entry labels.

For example, suppose our function `foo` uses all of the callee-save registers and needs three spill slots. Furthermore, suppose that at an internal label, `l`, there are two live variables, both of type `int4`, one in register `esi` and one in the middle spill slot. Then a correct pre-condition for `l` is:

```
l: ∀s1:Ts s2:Ts e1:Tcap e2:Tcap a1:T4 a2:T4 a3:T4.
 {esp:
      top4::int4::top4::a3::a2::a1
      ::{eax:int4
         esp: int4::s1@{esp:s2 eax:exn cap:e2}::s2
         ebp: {esp:s2 eax:exn cap:e2}::s2
         ebx:a1 esi:a2 edi:a3
         cap: &[e1,e2]}
      ::int4::s1@{esp:s2 eax:exn cap:e2}::s2}
  ebp: {esp:s2 eax:exn cap:e2}::s2
  cap: &[e1,e2]
  esi: int4}
```

Our register allocator tries not to use callee-save registers so that functions do not have to save and restore them. For example, suppose registers `esi` and `edi` are not used in a function. Then internal labels will encode that a value of type `a1` is on the stack in the appropriate place, `esi` contains a value of type `a2`, and `edi` contains a value of type `a3`.

If one or more records were partially initialized on entry to `l`, then the pre-condition would have a more complicated capability; we omit the details. What should be clear at this point is that the type annotations for internal labels are considerably less uniform than function-entry annotations.

4 Recovering Conciseness and Efficiency

Continuing the examples from the previous section, we describe three techniques for reducing the size of annotations. We then discuss techniques, most notably hash-consing, that can reduce the space and time required during verification. The next section quantifies the effectiveness of these and other techniques.

[6] It is theoretically possible to use polymorphism instead of a supertype, but in practice we found doing so very unwieldy.

4.1 Sharing Common Subterms

Because the annotations repeat information, we can greatly reduce their total size by replacing identical terms with a pointer to a shared term. As an example, consider again the pre-condition for the function `foo`, which takes and returns an `int`:

```
type exn = ∃c:Tm ^*[(^T^rw(c)*[int4^rw])^rw,c]
foo: ∀s1:Ts s2:Ts e1:Tcap e2:Tcap a1:T4 a2:T4 a3:T4 .
 {esp: {eax:int4
        esp: int4::s1@{esp:s2 eax:exn cap:e2}::s2
        ebp: {esp:s2 eax:exn cap:e2}::s2
        ebx:a1 esi:a2 edi:a3
        cap: &[e1,e2]}
       ::int4::s1@{esp:s2 eax:exn cap:e2}::s2}
  ebp: {esp:s2 eax:exn cap:e2}::s2
  ebx:a1 esi:a2 edi:a3
  cap: &[e1,e2]}
```

Removing some common subterms by hand, we can represent the same information with the following pseudo-annotation:

```
1 = ∃c:Tm ^*[(^T^rw(c)*[int4^rw])^rw,c]
2 = &[e1,e2]
3 = {esp:s2 eax: \boxed{1} cap:e2}::s2
4 = int4::s1@ \boxed{3}
5 = {eax:int4 esp: \boxed{4} ebp: \boxed{3}
     ebx:a1 esi:a2 edi:a3 cap: \boxed{2} }::\boxed{4}

foo: ∀s1:Ts s2:Ts e1:Tcap e2:Tcap a1:T4 a2:T4 a3:T4 .
 {esp: \boxed{5} ebp: \boxed{3} ebx:a1 esi:a2 edi:a3 cap: \boxed{2} }
```

Other pre-conditions can share subterms with this one. For example, the pre-condition for `l` from the previous section can be rewritten as:

```
l: ∀s1:Ts s2:Ts e1:Tcap e2:Tcap a1:T4 a2:T4 a3:T4.
 {esp: top4::int4::top4::a3::a2::a1::\boxed{5}
  ebp: \boxed{3} cap: \boxed{2} esi:int4}
```

Despite exploiting significant sharing, this example illustrates some limitations of sharing common subterms. First, we would like to share all the occurrences of "`s1:Ts s2:Ts ... a3:T4`", but whether or not we can do so depends on the abstract syntax of the language. Second, pre-conditions for functions with different parameter types or return types cannot exploit subterms 4 or 5. Another possible shortcoming not demonstrated is that alpha-equivalent terms may not appear to be the same. In practice, compilers can re-use variable names for compiler-introduced variables, so detecting alpha-equivalence for the purpose of sharing is not so important.

4.2 Parameterized Abbreviations

TALx86 provides user-defined (*i.e.* compiler-defined) higher-order type constructors. These functions from types to types have several uses. For example, they are necessary to encode source-level type constructors, such as array, list, or object types. Here we show how to use higher-order type constructors to define parameterized abbreviations. These abbreviations can exploit sharing among different types that sharing common subterms cannot. However, our verifier is unable to exploit such abbreviations during verification for reasons we explain below.

Because every function-entry pre-condition that our compiler creates is the same except for its parameter types and return type, we can create a parameterized abbreviation that describes the generic situation. Then at each function-entry label, we apply the abbreviation to the appropriate types.

```
type F = fn params:Ts ret:T4.
 ∀s1:Ts s2:Ts e1:Tcap e2:Tcap a1:T4 a2:T4 a3:T4.
 {esp: {eax: ret
        esp: params@s1@{esp:s2 eax:exn cap:e2}::s2
        ebp: {esp:s2 eax:exn cap:e2}::s2
        ebx:a1 esi:a2 edi:a3
        cap: &[e1,e2]}
       ::params@s1@{esp:s eax:exn cap:e2}::s2}
  ebp: {esp:s2 eax:exn cap:e2}::s2
  ebx:a1 esi:a2 edi:a3
  cap: &[e1,e2]}

foo: F int4::se int4
```

The only new feature other than the abbreviation is the type `se` which describes empty stacks. We use it here to terminate a list of parameter types. The use of abbreviations greatly simplifies the structure of the compiler because it centralizes invariants such as calling conventions.

It is not clear how a compiler-independent verifier could exploit an abbreviation like `F` during verification. Suppose the first instruction in block `foo` increments the input parameter. The verifier must check that given the pre-condition `F int4::se int4`, it is safe to perform an increment of the value on top of the stack. This verification requires inspecting the result of the abbreviation application — the verifier does not know that the argument `int4::se` describes the top of the stack. As we show in Section 5, using abbreviations sometimes slows down verification because of this phenomenon.

The abbreviation `F` is widely useful because all function-entry pre-conditions are similar. To use abbreviations for internal labels, we must capture the additional properties that distinguish these pre-conditions. In addition to `F`'s parameters, we also need parameters for the spill slots, the live registers, and something to do with partial-initialization issues. We also use a primitive type constructor (`&`) for combining two pre-conditions. That way we can pass in the live registers as one pre-condition and merge it with a pre-condition that describes the reserved registers.

```
type L =
fn params:Ts ret:T4 spills:Ts part:Tcap regs:Tpre.
 ∀s1:Ts s2:Ts e1:Tcap e2:Tcap a1:T4 a2:T4 a3:T4.
 {esp:
      spills@a3::a2::a1
      ::{eax: ret
         esp: params@s1@{esp:s2 eax:exn cap:e2}::s2
         ebp: {esp:s2 eax:exn cap:e2}::s2
         ebx:a1 esi:a2 edi:a3
         cap: &[e1,e2]}
      ::params@s1@{esp:s2 eax:exn cap:e2}::s2}
 ebp: {esp:s2 eax:exn cap:e2}::s2
 cap: &[part,e1,e2]}
 & regs

l: L int4::se int4
     top4::int4::top4::se ce {esi:int4}
```

L is correct, but it is useful only for labels in functions where all three callee-save values are stored on the stack. With a "don't optimize" approach, we could make all functions meet this description, but we lose most of the advantages of callee-save registers as a result. A better approach is to provide $2^3 = 8$ different abbreviations, one for each combination of callee-save values being stored on the stack. In fact, we need only 4 such abbreviations because our register allocator uses the callee-save registers in a fixed order. Because the compiler provides the abbreviations, this specialization is possible and appropriate.

Rather than require the compiler-writer to write and use higher-order abbreviations, one might hope to write a tool that took a collection of TALx86 types and re-wrote them in terms of some automatically generated abbreviations. Creating an optimal result appears at least as difficult as finding the shortest simply-typed lambda calculus term equivalent to a given one. We have not investigated using heuristics to discover useful abbreviations.

4.3 Eliding Pre-conditions

Recall that the verifier checks a code block by assuming its pre-condition is true and then processing each instruction in turn, checking it for safety and computing a pre-condition for the remainder of the block. At a control transfer to another block, it suffices to ensure that the current pre-condition implies the pre-condition on the destination label.

TALx86 uses a reconstruction approach by allowing many label pre-conditions to be elided. Clearly, the result of eliding a pre-condition is a direct decrease in annotation size. To check a control transfer to a block with an elided pre-condition, the verifier simply uses the current pre-condition at the source of the transfer to check the target block. Hence, if a block with elided pre-condition has multiple control-flow predecessors, it is verified multiple times under (possibly) different pre-conditions.

To ensure that the verifier terminates, we prohibit annotation-free loops in the control-flow graph. For this reason, TALx86 allows a pre-condition to be elided only if the block is only the target of forward jumps. Even with this restriction, the number of times a block is checked is the number of *paths* through the control-flow graph to the block such that no block on the path has an explicit pre-condition. This number can be exponential in the number of code blocks, so it is unwise to elide explicit pre-conditions indiscriminantly. As the next section demonstrates, an exponential number of paths is rare, but it does occur and it can have a disastrous effect on verification time.

The approach our compiler takes is to set an *elision threshold*, T, and insist that no code block is verified more than T times. Notice $T = 1$ means all merge points have explicit pre-conditions. We interpret $T = 0$ to mean that all code labels, even those with a single predecessor, have explicit pre-conditions. For higher values of T, we expect space requirements to decrease, but verification time to increase. Given a value for T, we might like to minimize the number of labels that have explicit pre-conditions. Unfortunately, we have proven that this problem is NP-Complete for $T \geq 3$. (We do not know the tractability when $T = 2$.) Currently, the compiler does a greedy depth-first traversal of the control-flow graph, leaving off pre-conditions until the threshold demands otherwise. In pathological cases, this heuristic can do arbitrarily poorly, but it seems to do well in practice.

Using an elision threshold is actually over-constraining the problem — it is more important to minimize the total number of times that we verify blocks. That is, we would prefer to verify some block more than T times in order to verify several other blocks many fewer times. For structured programs (all intra-procedural jumps are for loops and conditionals), it appears that this relaxed problem can be solved in polynomial time ($O(n^9)$ where n is the number of blocks [13]), but the algorithm seems impractical.

4.4 Hash-Consing and Fast Type Operations

So far, we have discussed techniques for reducing the size of the annotations that the code producer writes. For the verifier, these explicit types provide guidance to check that each assembly instruction is safe. To do this checking, the verifier determines the type of the context (i.e., the registers and the stack) before the instruction, the types of the operands, and the type of the context after the instruction. The operands must be subtypes of the types that the instruction requires. In short, the verifier itself creates many type expressions and often checks that one is a subtype of another. Therefore, it is important that these operations consume as little time and space as possible.

Our primary technique for reducing space is hash-consing, which is essentially just the on-line form of sharing. As types are created, we first check a table to see if they have been created before. If so, we return a pointer to the table entry;

if not, we put the type in the table. As a result, types consume less space, but we incur the overhead of managing a table. It would be correct to return any alpha-equivalent type from the table, but in the interest of fast lookup operations, we find only a syntactically identical type.

In the most general case, to decide if τ is a subtype of τ', we should convert both types to normal form and then do a structural subtyping comparison. One common case for which it is easy to optimize is when τ and τ' are the same object, that is, they are pointer-equal. With hash-consing, syntactically equal types should always be pointer-equal. Even when the two types are not the same object (for example, one is a strict subtype of the other), many parts of the two types may be pointer-equal, so we can usually avoid a full structural comparison.

There are complications with pointer equality, however: We must consider alpha-equivalent types to be equal. To do so, we maintain a separate variable-substitution map rather than actually performing costly type substitutions. In the presence of a non-empty map, it is not necessarily correct that pointer-equal types are equal because the substitution has not been applied. Fortunately, our compiler uses the same type variables consistently, so the variable-substitution map is almost always empty.

Hash-consing has another positive effect on verification time: When we reduce τ to τ' (for example, by applying an abbreviation), we do an in-place update of τ. Hence, all pointers to a shared τ will use the result of the single reduction. However, the original τ will no longer appear to be in the hash-cons table. We could add a level of indirection to alleviate this shortcoming (keep τ in the table for the purpose of future sharing and have it point to its reduced form τ'), but our implementation does not currently do so.

Another common operation on types is substitution, that is, substituting τ' for a variable α in τ. Operations that need substitution include applying abbreviations and instantiating polymorphic types. We need to recursively substitute for α in all the constituent types within τ, but we expect that most of them do not contain α. To optimize for this common case and avoid crawling over much of τ, we memoize the free variables of each type and store this set with the type.

As discussed in Section 5.4, we might expect further benefits from using de Bruijn indices and performing type substitutions lazily. Unfortunately, this change was so pervasive that we chose not to investigate it in the experiments that we discuss in the next section.

5 Experimental Results

In this section, we present our quantitative study of certifying a real program in TALx86. We conclude that targeting compiler-independent safety policies is practical and scalable when appropriate techniques are used.

Our example is the Popcorn compiler itself. The compiler consists of 39 Popcorn source files compiled separately. The more interesting optimizations performed are Chaitin-style intra-procedural register allocation [4] (using optimistic spilling [3] and conservative coalescing [8]) and the elimination of fully redundant null-checks for object dereferences. The entire compiler is roughly 18,000 lines of source code and compiles to 816 kilobytes of object code (335 kilobytes after running `strip`).

The sizes we report include the sum across files of all annotations, not just those for code labels. They do not include the separate module-interface files that the TALx86 link-checker uses to ensure type-safe linking. All execution times were measured on a 266MHz Pentium II with 64MB of RAM running Windows NT 4.0. The verifier and assembler are written in Objective Caml [15] and compiled to native code.

We first show that naive choices in the annotation language and compiler can produce a system with unacceptable space and/or time overhead. Then we show that our actual implementation avoids these pitfalls. Next we adjust various parameters and disable various techniques to discover the usefulness of individual approaches and how they interact. Finally, we discuss how we could extend our techniques to further lower the TALx86 overhead.

5.1 Two Bad Approaches

A simple encoding of the TALx86 annotations is insufficient. First, consider a system where we do not use the abbreviations developed in Section 4, our type annotations repeat types rather than share them, and we put types on all code labels. Then the total annotation size for our program is over 4.5 megabytes, several times the size of the object code. As for verification time, if we make no attempt to share common subterms created during verification, then it takes 59 seconds to verify all of the files.

A second possibility is to remove as many pre-conditions as possible. That is, we put an explicit pre-condition on a code label only if the label is used as a call destination, a backwards-branch destination, or a first-class value. Indeed, the total size of our annotations drops to 1.85 megabytes. However, the verifier now checks some code blocks a very large number of times. Total verification time rises to 18 minutes and 30 seconds.

These two coarse experiments yield some immediate conclusions. First, the actual amount of safety information describing a compiled program is large. Second, the number of loop-free paths through our application code is, in places, much larger than the size of the code. Therefore, it is unwise to make verification time proportional to the number of loop-free paths as the second approach does.

The latter conclusion is important for certified-code frameworks that construct verification conditions at verification time via a form of weakest pre-condition computation. Essentially, such systems construct pre-conditions for loop-free code segments using a backward flow analysis. In an expressive system, the pre-condition at a backward merge-point could be the logical disjunction of two conditions. Hence, if done naively, the constructed condition can have exponential size by having a different clause for every loop-free path.

When the number of loop-free paths is large, it is clear that constructing an enormous pre-condition is wasteful. For a compiler to exploit the weakness of such a pre-condition, it would need to have optimized based on an exponential amount of path-sensitive information. We conclude that constructing weakest pre-conditions in this way is impractical. Instead, annotations should guide the construction of the verification condition; the optional code pre-conditions of TALx86 fill this role.

5.2 A Usable System

Having shown how bad matters can get, we now present the actual overhead that our system achieves. First, we identify the main techniques used and the overhead that results. Then we show that verification time is roughly proportional to file size; this fact suggests that our approach should scale to larger applications. Finally, we partition the source code into several styles, show that the overhead is reasonable for all of them, and discuss salient differences.

Unlike the "straw man" systems constructed above, the real encoding of TALx86 annotations uses several tables to share common occurrences. Specifically, uses of identifiers, types, kinds, and coercions are actually indices into tables that contain the annotations. The code producer can avoid duplicates when constructing the tables. The benefit of this approach is proportional to the amount of repetition; there is a small penalty for annotations that occur only once. We call this technique "sharing"; more specifically it is full common-subexpression elimination on types at the file level. Sharing is just off-line hash-consing; we use the latter term to refer to sharing within the verifier for types created during verification.

Sharing does not create parameterized abbreviations, so we also use the abbreviations developed in Section 3. The compiler provides the abbreviations and uses them in a text version of TALx86. An independent tool converts the text version into a binary version that has sharing. In this sense, we use abbreviations "before" sharing.

We set the elision threshold to four. At this value, many forward control-flow points will not need explicit pre-conditions, but no block is verified more than four times.

Finally, the verifier uses hash-consing to share types that are created during verification. That is, when creating a new type, the verifier consults a table to see if it has encountered the type previously. If so, it uses the type in the table. Because the entire sharing table is parsed prior to verification, any types in the table will be used rather than repeated. Reductions on higher-order type constructors are performed in a lazy manner. In particular, we use a weak-head normalization strategy with memoization to avoid both unnecessary reductions and duplicated reductions. As such, other uses of the type will not have to recompute the reduction. Shao and associates use a similar strategy [26]. Because of complications with the scope of abbreviations, the hash-consing table is emptied before verifying each file. If memory becomes scarce, we could empty the table at any point, but this measure has not been necessary in practice. Note that the use of hash-consing cannot affect the size of explicit annotations; hash-consing attempts to share types created during verification.

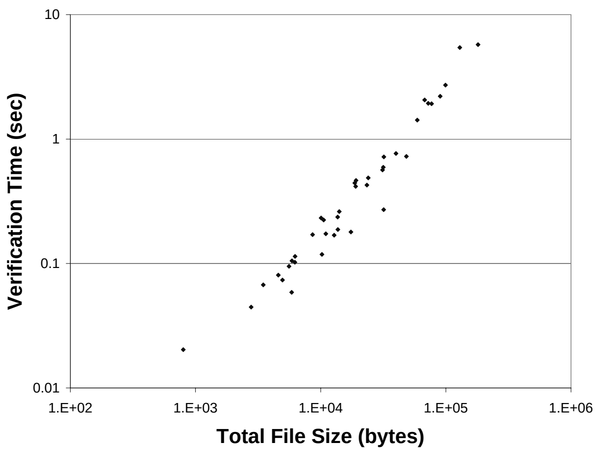


Fig. 1. Verification Time vs. File Size

With this system, total annotation size drops from 4.5 megabytes to 419 kilobytes and verification time drops from 59 to 34.5 seconds. As for compilation time, our compiler takes 40 seconds to compile the Popcorn source files into ASCII TALx86 files, which are essentially Microsoft Assembler (MASM) files augmented with annotations. A separate tool takes 23 seconds to assemble all of these files; this time includes the creation of the binary encoding of the annotations with sharing. As we add more optimizations to our compiler, we expect compilation time to increase more than verification time. The latter may actually decrease as object-code size decreases.

Performing `gzip` compression on the 419 kilobytes of annotations reduces their size to 163 kilobytes. The ratio of compression is similar to that for our object files; the unstripped files compress from 816 to 252 kilobytes and the stripped files compress from 335 to 102 kilobytes.

A desirable property is that verification time is generally proportional to file size. Without eliding pre-conditions, the time to verify TALx86 code is proportional to the size of the code plus the size of the *normalized* types used as annotations plus the time to look up types in the context. However, with higher-order abbreviations, normalizing types could, in theory, take non-elementary time [17]. We are pleased to see that such inefficiency has not occurred in practice: Figure 1 plots verification time against total size (object code plus annotations) for all of the files in the compiler. The time stays roughly proportional as file size grows by over an order of magnitude. Small files take proportionally longer to verify because of start-up costs and the overhead of using hash-consing. Such files take just a fraction of a second to verify, so we consider these costs insignificant.

So far we have presented results for the entire compiler as a whole. By analyzing the results for different styles of code, we can gain additional insight. Of course, all of the code is in the same source language, compiled by the same

Style	Object Code (kB)	Annotations (kB)	Verification Time (sec)	Size Ratio	Time Ratio
Polymorphic Libraries	36.4	19.6	1.19	.54	46.9
Monomorphic Libraries	34.8	15.1	.94	.43	53.1
Mostly Type Definitions	45.7	30.6	1.29	.67	58.9
Machine generated	148.4	82.0	6.30	.55	36.6
Compilation	550.0	271.4	22.3	.49	36.8

Fig. 2. Effect of Different Code Styles

compiler, and written by the authors. Nonetheless, we can partition the files into several broad categories:

- Polymorphic libraries: These files provide generally useful utilities such as iterators over generic container types. Examples include files for lists, dictionaries, sets, and resizing arrays.
- Monomorphic libraries: Examples include files for bit vectors and command-line arguments.
- Mostly Type Definitions: These files primarily define types used by the compiler and provide only simple code to create or destruct instances of the type. Examples include files for the abstract syntax of Popcorn, the compiler's intermediate language, an abstract syntax for TALx86, and an environment maintained while translating from the intermediate language to TALx86.
- Machine generated: These files include the scanner and the parser. Compared to other styles of code, they are characterized by a small number of large functions that contain switch statements with many cases. They also have large constant arrays.
- Compilation: These files actually do the compilation. Examples include files for type checking, register allocation, and printing the output.

Figure 2 summarizes the annotation size and verification time relative to the categorization.[7] The "Size Ratio" is annotation size divided by the object code size (smaller is better). The "Time Ratio" is the sum of the two sizes divided by the verification time (larger is better).

Most importantly, all of the size ratios are well within a factor of two and the time ratios are even closer to each other. We conclude that no particular style of code we have written dominates the overhead of producing provably safe object code. Even so, the results differ enough to make some interesting distinctions.

The files with mostly type definitions have the largest (worst) size ratio and largest (best) time ratio. The former is because type definitions are compiled into annotations that describe the corresponding TALx86 types, but there is no associated object code. The size ratio can actually be arbitrarily high as the

[7] The sum of the verification times is slightly less than the time to verify all the files together due to secondary effects.

		Annotation Size (kB)		Verification Time (sec)	
Sharing	Abbreviations	Uncompressed	Compressed	No hash-consing	Hash-consing
no	no	2041	155	50	38
no	yes	793	132	42	36
yes	no	503	205	37.5	34.5
yes	yes	419	163	40.5	34.5

Fig. 3. Effect of Abbreviations, Sharing Subterms, and Hash-Consing

amount of code in a source file goes to zero. The time ratio is also not surprising; the time-consuming part of verification is checking that each instruction is safe given its context.

The relatively high size ratio for machine-generated code is an artifact of how parsers are generated. Essentially, all of the different token types are put into a large union. The code that processes tokens is therefore filled with annotations that coerce values into and out of this union.

The size ratio for polymorphic libraries is slightly larger than we expected. A source-level function that is polymorphic over some types needs to explicitly name those types only once. Because TALx86 has no notion of function, all of the labels for such a function must enumerate their type variables.[8] Furthermore, control transfers between these labels must explicitly instantiate the additional type variables.

Finally, the time ratio is noticeably worse for the compilation code. This style of code contains a much higher proportion of function calls than libraries, which mostly contain leaf procedures. Because of the complicated type instantiations that occur at a call site, call instructions take the most time to verify.

5.3 Effectiveness of Individual Techniques

We have shown that our system achieves reasonable performance and uses a number of techniques for controlling annotation overhead, but we have not yet discussed which of the techniques are effective. In this section, we examine what happens if we selectively disable some of these techniques.

Figure 3 summarizes the total annotation size when the elision threshold is four and the other techniques are used selectively. When "Sharing" is no, we do not use tables for sharing types and coercions. Instead, we repeat the types directly in the annotations. We still share identifiers so that the lengths of strings is insignificant. If "Abbreviations" is no, then all abbreviations are fully expanded before the annotations are written. "Uncompressed" is the total size of all the annotations. "Compressed" is the sum of the result of running `gzip` on each file's annotations separately. The final two columns give total verification time with and without hash-consing enabled.

[8] Pre-conditions can still be elided, fortunately.

We first discuss the effect of sharing and abbreviations on the explicit annotation size. Both techniques appear very effective if we ignore the effect of `gzip`. Abbreviations alone reduce size by a factor of 2.57 whereas sharing alone reduces size by a factor of 4.06. Using abbreviations and sharing reduces size by another seventeen percent as compared to a system with just sharing. Hence neither technique subsumes the other, but they recover much of the same repetition.

However, if what we really care about is the size of annotations that must be sent to a code consumer, then we should consider running `gzip`. It is clear that `gzip` is extremely effective; our worst result for compressed annotations is a factor of two better than our best result for uncompressed annotations. More subtle is the fact that `gzip` achieves a smaller result when sharing is *not* used in our binary encoding. This result, which surprised us, is a product of how our tables are implemented and how `gzip` performs compression. In short, `gzip` constructs its own tables and uses a much more compact format than our encoding. Worse, our tables hide repetition from `gzip`, which looks for common strings. We conclude that if annotation size is the primary concern, then the binary encoding should remain "`gzip`-friendly".

Abbreviations are actually much more effective than the data in the figure suggests. The compiler's abbreviations are used only for code pre-conditions, so optimizing this one aspect of annotation size must eventually demonstrate Amdahl's Law.[9] We considered what the total annotation size would be if we removed *all* explicit code pre-conditions. Of course, the result of this drastic measure is unverifiable, but it provides a rough lower bound for the effectiveness of the abbreviations. The total size is still 377 kilobytes, so abbreviations reduced the size of code pre-conditions by about a factor of four $((2041-377)/(793-377))$.

We now discuss the effect of the techniques on verification time. Here `gzip` is useless because our verifier works on uncompressed annotations. Without hash-consing, sharing significantly reduces verification time. While the verifier under these conditions does not share types that it creates during verification, it does share types that originally occur in the annotations. The result suggests that these types cover many of those used during verification.[10] Without sharing, abbreviations are a great help because they recover the most common occurrences. However, with sharing, abbreviations actually *hurt* verification time. The time to expand the abbreviations during verification outweighs the time that the additional sharing gains.

With hash-consing, the different verification times are much closer to each other. Using a hash-consing table rediscovers any sharing, so without sharing initially we have to pay only the cost to achieve this rediscovery. More interestingly, the penalty for abbreviations disappears. We believe this result is due to the fact that with hash-consing, any abbreviation applied to the same argument is expanded only once and then the result is used in multiple places.

[9] Actually, there are a few other places where the abbreviations are used, such as when a polymorphic function is instantiated at a function type, but such situations are rare in our code.

[10] Parsing time is a small but noticeable fraction of the difference.

Hash-consing reduces verification time significantly, but only with a careful implementation of the hashing. For example, if we give our hash-cons table a size near a power of two (as number theory warns against), verification time takes longer than without hash-consing. The good news is that optimizing the verifier can sometimes be reduced to fundamental properties of data structures. The bad news is the difference between verification times under different parameters is more brittle than we would like.

One reason hash-consing improves verification time is that types occupy less space, so we expect better cache performance and fewer garbage collections. Another reason is that the verifier's function for determining if one type is a subtype of another returns immediately when two types are pointer-equal. This function is called about 170,000 times when verifying our compiler. Without hash-consing (but with sharing and abbreviations), 45,000 of the calls are with pointer-equal arguments. With hash-consing, the figure rises to 82,000. Even when the entire types are not pointer-equal, we can avoid much of the structural comparison when parts of them are pointer-equal. Without hash-consing, we make about 1,400,000 recursive calls. With hash-consing, the number of recursive calls drops to 730,000.

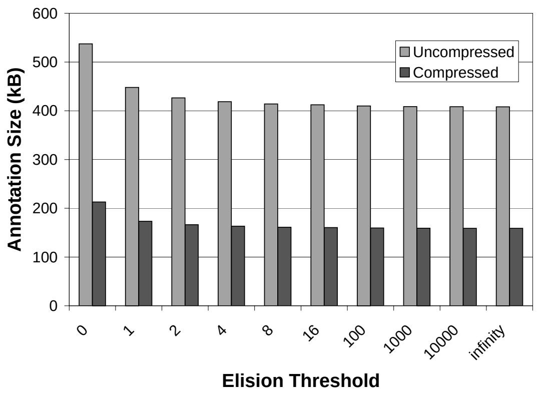
As explained in the previous section, TALx86 code blocks that are targets of only forward branches do not need annotations, but they will be reverified along every unannotated control-flow path. Given an elision threshold T, our compiler ensures that no block will be verified more than T times. Subject to this constraint, it uses a simple greedy algorithm to leave annotations off labels. Figure 4 shows the effect of changing the value of T. We use sharing and abbreviations.

The top chart in Figure 4 shows that total annotation size drops by over fifteen percent as T is 1 instead of 0. We conclude that low-level systems should not require pre-conditions on all blocks. However, the additional space savings as T takes values larger than 8 are quite small. This fact justifies the use of $T = 4$ for the other experiments.

The bottom charts in Figure 4 show the verification time for different values of T. Verification time initially drops as T gets the value 1 instead of 0. This phenomenon indicates that it takes a lot of time to process an explicit annotation and compare it to a pre-condition. As T takes values 2, 4, 8, and 16, verification time rises noticeably but only by a few seconds. We conclude that this range of values allows for reasonable time-space tradeoffs. As T takes larger values, verification time rises sharply. Although very few additional blocks have their pre-conditions elided, these blocks are then checked a very large number of times. In fact, for large T, the time spent verifying different files varies drastically because most files do not have any such blocks. (A value of infinity for T means we put explicit annotations only where the verifier requires them.)

5.4 Useful Extensions

We have presented a system where uncompressed safety annotations consume roughly half the space of the object code they describe, and we have given



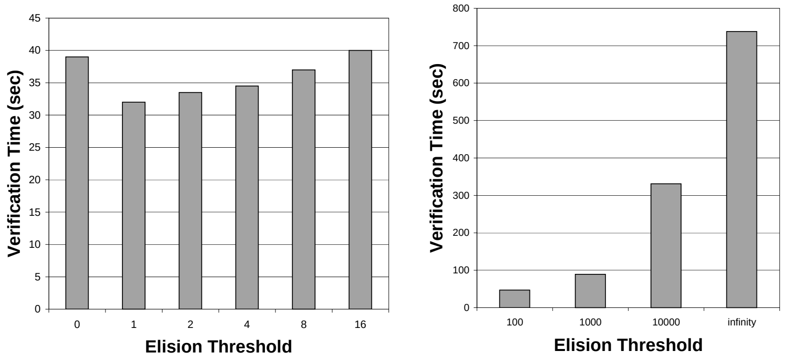


Fig. 4. Effect of Elision Threshold

techniques (sharing, abbreviations, and elision) that help in this regard. Now we investigate whether the current system is the best we can hope to achieve or if the techniques could contribute more to reducing the TALx86 overhead. By moving beyond what the current system supports, we demonstrate the latter.

First, notice that sharing common subterms is so effective because we share annotations across an entire file. The file level is currently the best we can do because we compile files separately. In a scenario where all of the object files are packaged together, we could share annotations in a single table for the entire package. Although our current tools cannot process such a package, we are able to generate it and measure its size. The total size drops from 419 kilobytes to 338 kilobytes. We conclude that different files in our project have many similar annotations; we should be able to exploit this property to further reduce overhead.

This improvement does not rely on understanding the compiler's conventions, so a generic TALx86 tool could put separately compiled object files into a package.

Second, the annotations that describe what coercions apply at each instruction are not currently shared. Although there are many common occurrences, some of them take only one byte to represent, so sharing these annotations must carefully avoid increasing space requirements.

Third, verification time suffers significantly from memory allocation and garbage collection. Although we have implemented hash-consing to address this bottleneck, Shao and associates [26] use their experience building type-directed compilers to suggest that suspension-based lambda encoding [21] (essentially de Bruijn indices and lazy substitution) can further improve performance. We relegate to future work modifying the verifier to experiment with these techniques.

Fourth, some well-chosen uses of type reconstruction could eliminate many of the explicit annotations. For example, if the verifier performed unification of (first-order) type variables, then the compiler could eliminate all of the type applications at control transfer points. This elision would improve our annotation size to 330 kilobytes. (To compute this figure, we elided the instantiations even though the verifier cannot process the result.) Reconstruction approaches improve the size even in the presence of `gzip`; the compressed annotations drop from 163 kilobytes to 141 kilobytes in this case.

In summary, the TALx86 system shows that techniques such as sharing and elision make certified code scalable and practical, but even TALx86 could use these techniques more aggressively to achieve lower overhead.

6 Conclusions

Our Popcorn compiler encodes the safety of its output in TALx86. As a Popcorn application itself, it also serves as the largest application we know of that has been compiled to a safe machine language. Because we believe safety policies should not be tailored to a particular compiler, we encode the aspects of Popcorn compilation relevant to safety in the more primitive constructs of TALx86. We have found that the most important factor in the scalability of certifying compilation is the size of code pre-conditions.

Based on our experience, we present the following conclusions for compiler-independent certification systems.

- Common-subexpression elimination of explicit annotations is a practical necessity. Sharing terms created during verification is also helpful, but it is important to carefully manage the overhead inherent in doing so.
- Compilers can effectively exploit parameterized abbreviations to encode their invariants. Although abbreviations improve the size of explicit annotations, it is more difficult to exploit abbreviations during verification.
- Serial compression utilities, such as `gzip`, are very helpful, but they are not a complete substitute for other techniques. Moreover, if good compression is a system requirement, one should understand the compression algorithm when designing the uncompressed format.

– Overhead should never be proportional to the number of loop-free control-flow paths in a program.

We believe these suggestions will help other projects avoid common pitfalls and focus on the important factors for achieving expressiveness and scalability.

Acknowledgments

The TALx86 infrastructure is a product of the TAL research group. Fred Smith contributed greatly to the prototype Popcorn compiler used for bootstrapping. We proved that optimal annotation elision is NP-complete for an elision threshold greater than five; David Kempe proved the other complexity results. The anonymous reviewers provided many helpful comments. David Walker generously encouraged us to write this report.

References

1. Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, MA, January 2000.
2. Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. Jazz: An efficient compressed format for Java archive files. In *CASCON'98*, November 1998.
3. Preston Briggs, Keith Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Progamming Languages and Systems*, 16(3):428–455, May 1994.
4. G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
5. Christopher Colby, Peter Lee, George Necula, and Fred Blau. A certifying compiler for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 95–107, Vancouver, Canada, 2000.
6. Karl Crary. A simple proof technique for certain parametricity results. In *Fourth ACM International Conference on Functional Programming*, pages 82–89, Paris, France, September 1999.
7. Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. Code compression. In *ACM Conference on Programming Language Design and Implementation*, pages 358–365, Las Vegas, NV, June 1997.
8. Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Progamming Languages and Systems*, 18(3):300–324, May 1996.
9. Neal Glew. Type dispatch for named hierarchical types. In *Fourth ACM International Conference on Functional Programming*, pages 172–182, Paris, France, September 1999.
10. Neal Glew and Greg Morrisett. Type safe linking and modular assembly language. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, TX, January 1999.
11. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
12. Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 60–74, San Antonio, TX, January 1999.

13. David Kempe. Personal communication.
14. Dexter Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, Ithaca, NY, January 1998.
15. Xavier Leroy. *The Objective Caml system, documentation, and user's guide*, 1998.
16. Steven Lucco. Split-stream dictionary program compression. In *ACM Conference on Programming Language Design and Implementation*, pages 27–34, Vancouver, Canada, June 2000.
17. Harry Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, September 1992.
18. 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 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, 1999. Published as INRIA Technical Report 0288, March, 1999.
19. Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, March 1998. Springer-Verlag.
20. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–569, May 1999.
21. Gapolan Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Duke University, Durham, NC, January 1994.
22. George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.
23. George Necula and Peter Lee. Efficient representation and validation of proofs. In *Thirteenth Symposium on Logic in Computer Science*, pages 93–104, Indianapolis, IN, June 1998.
24. George Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Twenty-Eigth ACM Symposium on Principles of Programming Languages*, pages 142–154, London, United Kingdom, January 2001.
25. William Pugh. Compressing Java class files. In *ACM Conference on Programming Language Design and Implementation*, pages 247–258, Atlanta, GA, May 1999.
26. Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Third ACM International Conference on Functional Programming*, pages 313–323, Baltimore, MD, September 1998.
27. Fred Smith, David Walker, and Greg Morrisett. Alias types. In *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Germany, March 2000. Springer-Verlag.
28. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.
29. Philip Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, London, United Kingdom, September 1989.
30. Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Progamming Languages and Systems*, 19(1):87–152, January 1997.