Scalable Certification of Native Code: Experience from Compiling to TALx86

Dan Grossman Greg Morrisett *

Department of Computer Science
Cornell University

Abstract

Certifying compilation allows a compiler to produce annotations that prove that target code abides by a specified safety policy. An independent verifier can check the code without needing to trust the compiler. For such a system to be generally useful, the safety policy should be expressive enough to allow different compilers to effectively produce certifiable code.

In this work, we use our experience in writing a certifying compiler to suggest general design principles that should allow concise yet expressive certificates. As an extended example, we present our compiler's translation of the control flow of Popcorn, a high-level language with function pointers and exception handlers, to TALx86, a typed assembly language with registers, a stack, memory, and code blocks. This example motivates techniques for controlling certificate size and verification time.

We quantify the effectiveness of techniques for reducing the overhead of certifying compilation by measuring the effects their use has on a real Popcorn application, the compiler itself. The selective use of these techniques, which include common-subexpression elimination of types, higher-order type abbreviations, and selective re-verification, can change certificate size and verification time by well over an order of magnitude. We consider this report to be the first quantitative study on the practicality of certifying a real program using a type system not specifically designed for the compiler or source language.

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 typing annotations that a dataflow analysis-based verifier can use to ensure that the code is type safe.

However, both the instructions and the type system of JVML are 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 method-return operations. Also, it provides no provision for performing general tail-calls on methods. Therefore, JVML cannot be used as a target for certifying compilers of functional programming 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 certifying. That way we can verify the safety of the resulting code instead of trusting a large optimizing compiler.

Another example of a certifying compiler is Necula and Lee's Touchstone compiler [18]. Touchstone compiles a small, type-safe subset of C to high-performance 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 when compared to the *ad hoc* verification process of the 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 [8]. 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.

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.

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 implementa-

*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.

tion of general-purpose type systems suitable for assembly language [15, 16, 14]. 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 the Intel IA32 assembly language and a verifier that we call TALx86 [13]. Where possible, we have avoided baking in 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 did not have to change the type system of TALx86 to support these optimizations. In this respect, 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 typing annotations — *often orders of magnitude larger than the code itself*. Thus, there is a very real danger that our noble goal of using one general-purpose type system will be defeated by practical considerations of space and time.

Some researchers, such as Appel and Felty [1], suggest going even further than we do with respect to minimizing potential soundness errors. In particular, they have recently proposed formally specifying the type system in terms that relate directly to the underlying machine semantics. By embedding the policy in a higher-order logical framework, they hope that compilers may define their own type systems and formally prove them sound with respect to the underlying semantics. This work is a promising direction for developing an infrastructure that is truly independent of the source language and compiler. However, it is clear that an even more primitive approach than ours will only exacerbate the problems with respect to certificate size and verification time.

The work presented here is essentially 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 easily ported various ML libraries to Popcorn and constructed the certifying compiler for Popcorn in Popcorn itself.

Although the TALx86 type system is very expressive, it is far from a universal typed assembly language. 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.

In the next section, we give a taxonomy for general approaches to reducing typing annotation overhead and further discuss other projects related to certifying compilation. Then 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, where we conclude that the TALx86 approach scales to a medium-size application and that almost all of the techniques contribute significantly to reducing the overheads 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 short 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 only supports 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. Both the `javac` and Touchstone compilers 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.

Even very general frameworks inevitably bake in more than the underlying machine requires. Although the work by Appel and Felty aims to remove all such restrictions, doing so soundly is extremely difficult. For example, their current system does not allow a memory location to be written twice. This restriction not only effectively limits the current applicability to purely functional source languages, but it inhibits implementation strategies as common as using a call stack.

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 simply 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 [10]. 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 [24]. However, the optimized code requires a more sophisticated type system to convince the verifier that type tests are unnecessary. In general, such type systems require additional annotations to make verification tractable. For example, the Touchstone type system supports static elimination of array bounds checks, but requires additional invariants and proof terms to support the optimization.

Another common 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 which 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 [19].

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.

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. 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. Since the verifier is aware of this sharing, it can exploit it to consume less space. There is an interesting trade-off 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[20, 2]. For example, these 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 [21] 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 [17] as a solution. 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.

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 [23]. As we show in Section 3, 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 since she is already reasoning in terms of these abstractions. However, if the verifier must expand the abbreviations in order to verify the code, no gain in verification space is achieved and verification time may suffer.

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`, Kozen's ECC, or Touchstone, 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. Also, memory locations are statically divided into code and data

(although extensions support run-time code generation[9]). In order to investigate the practicality of expressive low-level safety policies, we have relied on a rigorous, informal proof of type soundness and a procedural implementation of the verifier.

This approach has allowed us to examine the feasibility of compiler-independent safety policies on a far larger scale than has been previously possible. At the time of this writing, no compiler has targeted the independent safety policies of Appel and Felty. The published results of the Touchstone project, which does not have a compiler-independent safety policy, do not include object files larger than four kilobytes [18]. More recent work on a Java-specific safety policy has so far compiled only very small programs [11]. In contrast, the data we present in Section 5 is the result of compiling all 39 modules of a real program. The result of compilation is hundreds of kilobytes of object code that link together to form an executable program.

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. More thorough discussions of the underlying formalism may be found elsewhere [16, 14]. Here, we concentrate on the details relevant to our study.

A TALx86 object file consists of Intel 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 typing annotation. The typing annotations on the labels of instruction sequences, called *code types*, specify a typing 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:^*[int4,int4]}`, then control may only be transferred to the address denoted by `L` 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 data-flow 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 an isomorphic type or to explicitly instantiate polymorphic type variables.

Not all labels require a typing annotation. 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,[1] or data structure (*e.g.* exception handlers); on labels that are the targets of backwards branches (*e.g.* loop headers); and on labels that are exported from the object file module (*e.g.* function entry points.) These restrictions ensure that verification terminates.

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

- An intra-procedural jump (e.g., loops and conditionals)
- A function call or return
- An invocation of the current exception handler

Currently, our compiler only performs 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.)

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 an intuitive feeling for the following claims which we believe transcend TALx86:

- If the safety policy does not bake in data and control flow abstractions, then the annotations 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 less so in the presence of optimizations.

Because of this focus, we purposely do not explain some of the aspects of the annotations other than to mention the general things they are encoding. The reader interested in such details should consult the literature[16, 14, 7, 13, 6, 22].

3.1 Function Entry Labels

As a running example, we consider a Popcorn function `foo` which takes one parameter, an `int`, and returns an `int`. The Popcorn type `int` is compiled to the TALx86 type `int4`. Arithmetic operations are allowed on values of this type, but treating them as pointers is not. Our compiler uses the standard C calling convention for the IA32. Under this convention, the parameters are passed on the stack, the caller pops the parameters, the return address is shallowest on the stack, and the return value is passed in register EAX. 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}
```

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

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 forall-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.

Our annotation does not quite describe the standard C calling convention. In particular, the standard requires 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:[2]

```
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`,[3] the appropriate registers must have those types before `foo` is called and again when the return address is invoked. This annotation restricts the behavior of `foo` to preserve these registers since it does not know of any other values with these types. Notice that if we wish to use different conventions about which registers should be callee-saves, then we need only change 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, 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. Since `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 we use `@` 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}
```

[2] Here and below, underlining is for emphasis; it is not part of TALx86.

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

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[6]. Each of the four occurrences of `exn` above should in fact be replaced by the typing expression

```
∃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 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*[22], as shown below, to indicate that it does not create any aliases to partially initialized records held by the caller and exception handler.

```
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 which 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. Moreover, even these parts are the same for all functions that take and return integer values.

3.2 Internal Labels

In this section, we present the pre-conditions for labels that are only targets of intra-procedural jumps. For simplicity, we only discuss labels in 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 be stored 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 "placeholder" 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 single type can serve as this placeholder. TALx86 provides a primitive type `top4` which is a super-type of all types of kind `T4`. We give this type to the dead stack slots at the control-flow join and the appropriate subtyping on control transfers is handled implicitly by the verifier.

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 a given internal label are considerably less uniform than function entry point annotations.

4 Recovering Conciseness

Continuing the examples from the previous section, we describe three techniques for reducing the size of annotations. The next section quantifies the effectiveness of these and other techniques.

4.1 Sharing Common Subterms

Since the annotations repeat information, we can greatly reduce their total size by replacing identical terms with a pointer to a shared term. This technique is common referred to as "hash consing". 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: 1 cap:e2}::s2
4 = int4::s1@ 3
5 = {eax:int4 esp: 4 ebp: 3
     ebx:a1 esi:a2 edi:a3 cap: 2 }::4

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

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

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

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. Such abbreviations can be used to share common patterns that hash-consing of subterms cannot, but for our verifier, it is difficult to exploit abbreviations during verification.

Since every function entry pre-condition created by Popcorn is the same except for its parameter types and return types, 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 since the verifier does not know that the argument `int4::se` describes the top of the stack. As we show in the section on experimental results, using abbreviations sometimes slows down verification because of this phenomenon.

The abbreviation `F` can be used often 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@{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@{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 only useful 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 only need four such abbreviations since our register allocator uses the callee-save registers in a fixed order. Since the compiler is the creator of the abbreviations, this specialization is possible and appropriate.

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 takes 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 termination of verification, we must 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 disastrous effects on verification time.

The approach taken by our compiler 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, this problem is NP-Complete for $T \geq 3$. Currently, the compiler does a simple greedy depth-first traversal of the control-flow graph, leaving off pre-conditions until the threshold demands otherwise.

5 Experimental Results

In this section, we present our quantitative study of certifying a real program in TALx86. We conclude that tar-

geting compiler-independent safety policies is practical and scalable, but that careful use of appropriate techniques can significantly reduce verification space and time.

Our example is the Popcorn compiler itself. This choice saves us the effort of writing a compiler and an application. It also suggests that our application is useful and our source language is realistic. Of course, studying a new application written by ourselves in our own language makes many comparisons impossible. Since we are more interested in studying the overheads of compiling to expressive low-level safety policies than in producing excellent code, we find this approach attractive.

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[5]) 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`).

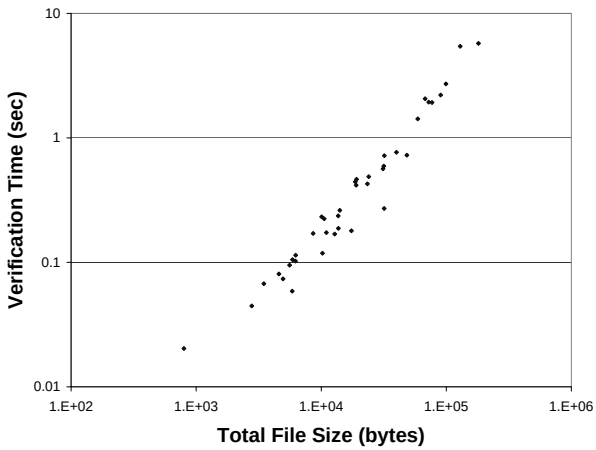
The sizes we report include the sum across files of all annotations, not just those for code labels. They do not include interface information used for type-safe linking. All execution times were measured on a 266MHz Pentium II with 64MB of RAM running Windows NT 4.0. The verifier and assembler are written in Objective Caml[12] 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 sharing 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 only put an explicit pre-condition on a code label 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 (such as proof-carrying code) that construct verification conditions at verification time via a form of weakest