Type-Preserving Compilation for Large-Scale Optimizing Object-Oriented Compilers *

Juan Chen Chris Hawblitzel
Microsoft Research
{juanchen, chrishaw}@microsoft.com

Frances Perry
Princeton University
frances@cs.princeton.edu

Mike Emmi
University of California, Los Angeles
mje@cs.ucla.edu

Jeremy Condit Derrick Coetzee
Microsoft Research
{jcondit, dcoetzee}@microsoft.com

Polyvios Pratikakis
University of Maryland, College Park
polyvios@cs.umd.edu

Abstract

Type-preserving compilers translate well-typed source code, such as Java or C#, into verifiable target code, such as typed assembly language or proof-carrying code. This paper presents the implementation of type-preserving compilation in a complex, large-scale optimizing compiler. Compared to prior work, this implementation supports extensive optimizations, and it verifies a large portion of the interface between the compiler and the runtime system. This paper demonstrates the practicality of type-preserving compilation in complex optimizing compilers: the generated typed assembly language is only 2.3% slower than the base compiler's generated untyped assembly language, and the type-preserving compiler is 82.8% slower than the base compiler.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Verification

Keywords Type-preserving compilation, object-oriented compilers

1. Introduction

Because of compiler bugs, compilers may not preserve safety properties of source-level programs through the compilation process. Over the last decade, many researchers have proposed techniques for removing the compiler from the trusted computing base by ensuring that the output of the compiler has the same safety properties as the input. Necula and Lee proposed Proof-Carrying Code (PCC), in which low-level code is accompanied by a safety proof that can be verified efficiently [12]. Morrisett *et al.* developed Typed Assembly Language (TAL), in which the compiler produces type-annotated assembly code; here, the soundness of the TAL type system guarantees that a well-typed TAL program is both type-safe and memory-safe [11]. Leroy built a certified compiler with a formal proof that the compilation preserves the semantics of source programs [8]. However, none of these compilers was a large-scale optimizing compiler that is designed for use in real-world software development. Such compilers may be orders of magnitude larger in size than research prototypes because of their support for advanced language features as well as their aggressive optimization.

This paper presents the implementation of type-preserving compilation in a large-scale optimizing compiler of approximately 200,000 lines. By type-preserving compilation, we mean that the compiler preserves types through each intermediate representation, from source code to assembly code, allowing a lightweight verifier to check the safety of the generated assembly code without trusting the compiler itself. Our compiler is about an order of magnitude larger than previously published type-preserving compilers,[1] and it is used by about 30 developers on a daily basis. Although the compiler was originally implemented with a typed intermediate representation, these types were previously discarded prior to instruction selection and register allocation. With our enhancements, these types are preserved down to the assembly code, where they can be checked by a standalone verifier.

The source language of the compiler is Microsoft Common Intermediate Language (CIL) [6]. We chose a compiler for an object-oriented language because object-oriented languages such as Java, C#, and C++ are among the most popular programming languages for industrial software development. For our medium-level and low-level type system, we chose to implement LILC (Low-level Intermediate Language with Classes) [2] and SST (Simple Stack Types) [13] because they are sound and have decidable type checking. In addition, these type systems are expressive enough to support *type-safe* implementations of language features such as dynamic dispatch and reference parameters.

Our work makes the following contributions:

1. We implemented the LILC and SST type systems in an existing optimizing compiler at reasonable cost. Although these type systems were published previously, they had not been implemented or tested in a practical setting. Our compiler uses approximately 40 low-level optimizations, and with our changes

* The work by Frances Perry, Mike Emmi, and Polyvios Pratikakis was done during internship at Microsoft Research

[1] SpecialJ has about 33k lines for the compiler and 25k for the VCGen and the checker [4]. The TALx86 compiler has about 18k lines [7]. The LTAL compiler has about 50k for the typed backend [3].

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-59593-860-2/08/06...$5.00

to the compiler, it can preserve types through almost all of these optimizations. For the benchmarks we measured (ranging from 4MB to 20MB in executable size), the generated TAL code is 0.95-1.09 times slower than the base compiler's generated code, with a geometric mean of 1.02. The TAL compiler is 0.99-17.12 times slower than the base compiler, with a geometric mean of 1.83. The TAL checking time is 1.61%-18.78% of the compilation time, with a geometric mean of 5.59%. The TAL size is 1.69-2.52 times larger than the x86 code in terms of object file size, with a geometric mean of 2.05. We achieved this result by modifying approximately 10% of the existing compiler code (19,000 out of 200,000 lines) and by adding 12,000 lines of code for TAL definition and verification.

2. The TAL compiler generates explicit proofs about integer values, and the TAL checker verifies these proofs. In particular, the compiler generates proofs that array indices are within array bounds, even for cases where the compiler's optimizations eliminate run-time bounds checks. The compiler generates the necessary loop invariants and proofs just before converting to the back-end intermediate representation, and it preserves the invariants and proofs through the back-end phases.
3. The TAL checker verifies a large portion of the interface between the compiler and the runtime system. The current system trusts the implementation of the runtime system, including the garbage collector, but it verifies most of the object layout and garbage collection information that is generated by the compiler for use by the runtime system.

We believe that type-preserving compilation is a practical way to verify the output of large-scale optimizing compilers. Types are a concise way to represent data properties, and they impose few constraints on optimizations. Because many existing compilers already use types and type checking internally, it is natural to extend these compilers to preserve types at a low level.

These benefits give type-preserving compilation a number of advantages over using a certified compiler, where the compiler itself is proven correct once and for all. Whereas existing optimizing compilers can be extended to preserve types in a natural way, it is not obvious whether we can certify a large-scale optimizing compiler. For example, Leroy's work [8] showed that well-understood transformations such as layout of stack frame are difficult to prove correct, and a production compiler will have many such analyses and transformations. Furthermore, a realistic compiler is under constant change, which requires the compiler to be re-proven after each change.

The rest of the paper is organized as follows. Section 2 provides background information regarding the base compiler and our type systems. Section 3 presents our implementation of the type-preserving compiler. Section 4 discusses our techniques for verifying array bounds accesses, Section 5 presents our approach to verifying interaction with the garbage collector, and Section 6 discusses our performance results. Section 7 discusses related work, and Section 8 concludes.

2. Background

2.1 The Base Compiler

The base of our implementation is Bartok, which allows the use of managed languages like C# for general-purpose programming. The compiler has about 200,000 lines of code, mostly written in C#, and is fully self-hosting.

Bartok can compile each binary module individually or it can compile them all as a whole program with interprocedural optimization. We use separate compilation mode to separate user programs from the libraries that we currently do not verify.

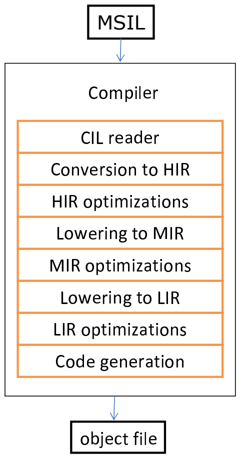

Figure 1. Architecture of Bartok

Performance of Bartok's generated code is comparable to their performance under the Microsoft Common Language Runtime (CLR). According to the benchmarks tested, programs compiled by Bartok are 0.94 to 4.13 times faster than the CLR versions, with a geometric mean of 1.66.

The architecture of Bartok is shown in Figure 1. Bartok translates programs written in CIL, which is an intermediate representation for C# and other languages, into native x86 code, through three intermediate representations: High-level IR (HIR), Medium-level IR (MIR), and Low-level IR (LIR). Bartok performs about 40 optimizations on the three IRs.

HIR is similar to CIL, except that HIR does not use stack-based computation. The type system of HIR is almost the same as that of CIL (or C#), consisting of primitive types, classes, interfaces, arrays, and so on. All object-oriented operations are primitives in HIR, such as virtual method invocation, type cast, and array access.

These primitives are lowered during the translation from HIR to MIR. For example, virtual method invocation is translated to code for fetching the vtable out of an object, fetching a method out of the vtable, and calling the method on the object. The original MIR had the same type system as HIR, which is not expressive enough to represent the result of such a lowering. For example, no HIR types could represent the vtable of a class that contains virtual methods.

MIR is further lowered to LIR. Then several transformations are performed on LIR: class layout, instruction selection, register allocation, and stack frame layout. The original LIR and the back end were untyped.

The lowest level of LIR (essentially assembly) is written to object files in a standard format. A standard linker links the object files and creates native x86 executables.

2.2 The LILC Type System

LILC is a typed intermediate language we implement as the typed MIR in Bartok to represent implementations of lowering object-oriented primitives.

LILC was developed by Chen and Tarditi for compiling core object-oriented language features, such as classes, objects, and arrays [2]. We decided to use LILC because it faithfully represents standard implementations of object layout, virtual method invocation, and runtime libraries such as type test and array store check. Furthermore, LILC preserves notions of classes, objects, and subclassing, whereas other encodings compile those notions to functional idioms (records and functions). It is easier to implement LILC in Bartok because LILC preserves those object-oriented no-

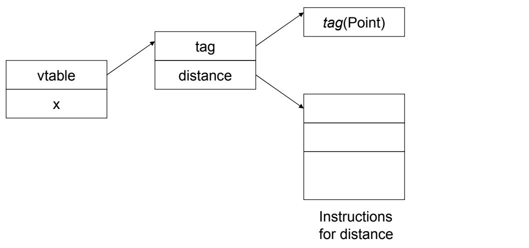

Figure 2. Object Layout for Point

tions in the input language of Bartok. LILC has been proven sound and its type checking is decidable.

This section explains the main ideas of how LILC represents objects and runtime libraries. Other features can be found in earlier work [2].

Classes and Objects LILC has both nominal (name-based) class names and structural record types. Each class C has a corresponding record type $R(C)$ that describes the object layout for the class C. For a typical layout strategy, an object contains a vtable and a set of fields. The vtable contains a tag—a unique identifier to identify the class at runtime, and a set of virtual methods.

Suppose a class Point is defined as follows:

class Point {
  int x;
  virtual int $distance()\{\ldots\}\}$

The class Point contains an integer field x and a virtual method $distance$ that takes no parameters and returns an integer. The object layout of Point is shown in Figure 2.

Type R(Point) represents this layout naturally:

$$\begin{aligned}&R(\text{Point}) =\\&\quad\{vtable : \{tag : \text{Tag(Point)},\\&\qquad\qquad distance : (\exists\alpha \ll \text{Point}.\ \alpha) \rightarrow \text{int}\},\\&\quad x : \text{int}\}\end{aligned}$$

R(Point) is a record type with two fields, $vtable$ and x. The type for $vtable$ is another record type containing fields tag and $distance$. Note that objects and pointers to objects are used interchangeably in this paper. It should be clear from the context to which we refer. For example, the vtable field is actually a pointer to a record of two fields, but we omit the representation of the pointer in the type and use a record type directly.

The tag of the class Point identifies Point at run time. Its type is represented as Tag(Point), where Tag is an abstract type constructor. LILC treats tags as abstract for simplicity.

The vtable contains a method pointer for the virtual method "distance". The method "distance" now takes one parameter— the "this" pointer—to reflect the self-application semantics where we pass an object to a virtual method when calling the method on the object. The type of the "this" pointer requires that the "this" pointer be an instance of Point or Point's subclasses. We explain "this" pointer types later in this section.

An instance of C can be coerced to and from a record of type $R(C)$ without any runtime overhead. The coercions are runtime no-ops. Objects are lightweight because interesting operations are performed on records. Object and class notions are preserved to simplify the type system.

The separation of the nominal and structural types eliminates explicit structural recursion, because the record type $R(C)$ can refer to any class name, including C itself. Also, the separation allows a straightforward typing of self-application semantics, which is the most challenging problem in typed intermediate languages for object-oriented languages.

Virtual Method Invocation Virtual method invocation requires a distinction between the static type and the dynamic type (actual runtime type) of an object. To call a method m on an object o with static type C, we need to pass o as the "this" pointer to m, or at least pass an object that has the same *dynamic* type (or its subclasses) as o. Passing an object with the same static type C may be unsafe.

Consider the following example:

void Unsafe(Point p, Point q){
  $vt = p.vtable;$
  $dist = vt.distance;$
  $dist(q);\}$

This function is unsafe, even though the distance method fetched from p requires an object of Point and q is indeed an object of Point. The function can be called in an unsafe way if p is actually an instance of a subclass of Point and the subclass overrides $distance$ to access fields in the subclass but not in Point.

To guarantee the soundness of virtual method invocation, LILC introduces "exact" notions of classes to represent dynamic types. Unlike source languages Java and C# where a class name C represents objects of C and C's subclasses, LILC uses C to represent only objects of "exact" C, not C's subclasses. LILC uses an existential type $\exists\alpha \ll C.\ \alpha$ for objects of C and C's subclasses. The notion "$\ll$" represents subclassing. The type variable α indicates the dynamic type, which must be a subclass of C. Objects with source-level type (or static type) C are translated to have the existential type in LILC. LILC has subtyping rules to guarantee that subclass objects can be used as superclass objects.

Suppose an object has dynamic type τ. Any virtual method fetched from the object has "this" pointer type $\exists\alpha \ll \tau.\ \alpha$, meaning only objects of τ or τ's subclasses.

The "Unsafe" example is ill-typed in LILC. The object p has type $\exists\alpha \ll$ Point. α in LILC. To invoke method "distance" on p, we first open p and introduce a type variable β for p's dynamic type. The type of the "distance" method fetched from p requires that the "this" pointer be an object of β or β's subclasses. The type checker accepts passing p to "distance" but rejects passing q because we cannot guarantee and q is an object of β or β's subclasses.

Type-safe Runtime Libraries The type-safe runtime libraries are represented in the LILC type system exactly like user programs. This gives the compiler freedom to inline and optimize them.

Type Cast. Downward type casts check at run time whether an arbitrary object is an instance of an arbitrary class or its subclasses. In a typical implementation, each class stores a tag in its vtable. If C extends B, then the tag of C has a pointer pointing to the tag of B. The pointers form a tag chain. Downward type cast fetches tags in the chain and compares them with the tag of the target class.

This typical implementation is expressed as a well-typed polymorphic function in LILC that can be applied to arbitrary objects and arbitrary classes. The key ideas are to use types to connect an object with the tag it contains, and to refine types according to the tag comparison result. Two classes are the same if and only if their tags are equal. If an object o has type τ and the tag in o is equal to $tag(C)$, then $\tau = C$. If one of the parent tags, which identifies a parent class of τ, is equal to $tag(C)$, then $\tau \ll C$ and o can be cast to C.

Array Store Check. Source languages such as Java and C# have covariant array types, that is, if A is a subtype of B, then $Array(A)$ is a subtype of $Array(B)$. Covariant array types require runtime "store checks" each time an object is stored into an array. If array a has static type array(B), to store an object of type B in a, we have to check whether the object has the "actual" element type of a because a might be an array of A.

LILC uses invariant array types enclosed with existential types to express source-level array types. An LILC array type is a subtype of only itself. The source-level array subtyping is transferred to subtyping on the enclosing existential types in LILC.

To store an object in an array that has type array(C), LILC programs must explicitly check whether the object is an instance of the element type C, which can utilize the previous type cast function.

2.3 The SST Type System

SST is a simple, sound, and decidable type system that supports most common stack operations, aliased stack locations, and by-reference parameters [13]. Other type systems for stacks either were undecidable or did not support by-reference parameters.

A stack is viewed as a sequence of stack slots. The stack grows toward lower addresses. The stack can be accessed by arbitrary pointers to the stack. But only changing the value of the stack pointer "sp" can grow or shrink the stack.

SST uses a stack type to describe the current stack state. SST checks one function at a time. Each function can see only its own stack frame. The previous stack frames are abstracted by a stack variable.

Each stack slot in the current stack frame is labeled by a location. A pointer to a stack slot labeled by location ℓ has a singleton type Ptr(ℓ). The stack type tracks mapping from stack locations to current types of values stored at those locations. We call such mapping *capabilities*. Capabilities are linear, that is, they cannot be duplicated. Because of the separation between singleton pointer types and capabilities, the capabilities can evolve, independently of the pointer types, to track updates and deallocation.

SST uses a non-commutative, non-associative operator "::" to glue capabilities together to form a stack type: "ℓ_2 : int :: ℓ_1 : int :: ℓ_0 : ρ" means that two integers are on top of the stack (at locations ℓ_1 and ℓ_2) and the rest of the stack is abstracted as a stack variable ρ. Capabilities glued by :: form the spine of a stack.

To represent aliasing, SST introduces a "$\wedge$" operator to attach a capability to a stack type. The capability describes an aliased location to some location inside the stack type. Therefore, the scope of the capability is the stack type: the capability is safe to use as long as the stack type is not modified. To guarantee safety, the scope can be only expanded to a larger stack type but not contracted.

By-reference Parameters. Consider a function $swap$ that has two by-reference integer parameters x and y:

```
void swap(ref int x, ref int y) {...}
```

Suppose the compiler pushes parameters onto the stack from right to left. In SST, the stack type in the precondition of $swap$ is next(next(ℓ_0)) : Ptr(ℓ_x) :: next(ℓ_0) : Ptr(ℓ_y) :: ℓ_0 : $(\rho \wedge \{\ell_y : \text{int}\} \wedge \{\ell_x : \text{int}\})$.

Upon entry to $swap$, the stack holds the arguments x and y on its top, each of which is a pointer to some aliased location inside ρ. Note that aliased locations ℓ_x and ℓ_y may appear anywhere in ρ, in any order. In fact, ℓ_x and ℓ_y may be the same location.

The $swap$ function can pass x and y to other functions, further expanding the scopes of ℓ_x and ℓ_y. But $swap$ cannot return a reference to a local variable defined in itself, because this contracts the scope of the reference to ρ.

2.4 Arrays

Loads and stores to array elements are primitive operations in CIL. Naively, each array element access requires a run-time bounds check to ensure that the element index is within the array's bounds. Compilers break up each array access into a sequence of more primitive machine instructions. Many compilers, including Bartok, will also optimize away many of the bounds checks. The TAL type system must be able to ensure that every access is within bounds even after the compiler's transformations and optimizations.

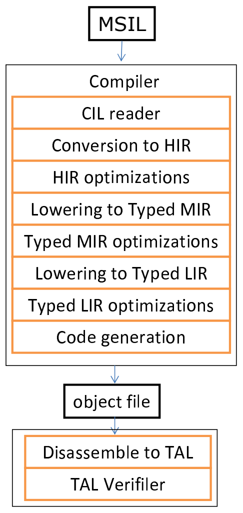

Figure 3. Architecture of the New Compiler

To solve this problem, our type system incorporates techniques from earlier work. First, following Xi and Harper's work on DTAL [16], it uses singleton types to represent known information about integer values. For example, the number 5 has type int, but 5 also has the more specific type $S(5)$, the singleton type of integers equal to 5. More interestingly, a variable x might have singleton type $S(\alpha)$, where a basic block's pre-condition specifies a constraint $\alpha < 5$ on the integer type variable α. The singleton type together with the constraint ensure that x is less than 5. DTAL used an arithmetic constraint solver to verify that the pre-conditions for basic blocks and array operations are satisfied. Unfortunately, the constraint solver sits in the TAL verifier's trusted computing base, and for decidability's sake the solver is limited to a subset of arithmetic. Therefore, our type system follows the approach advocated by Crary and Vanderwaart [5] and Shao *et al.* [14], which keeps DTAL's singleton types and pre-conditions, but also allows explicit proofs of pre-conditions in the TAL program. In this approach, the TAL verifier only needs to check the supplied proofs, and does not need to automatically solve arithmetic constraints. Of course, the compiler must generate the proofs before the TAL verifier can check the proofs; proof generation for arrays was beyond the scope of Crary and Vanderwaart [5] and Shao *et al.* [14]. Section 4 describes how Bartok generates these proofs.

3. Implementation of a Type-Preserving Compiler

This section provides implementation details for the new type-preserving compiler. In order to preserve types, we must implement typed MIR and LIR using the LILC and SST type systems. We must also ensure that types are preserved across low-level optimizations.

The architecture of the new compiler is shown in Figure 3. The original MIR type system was enhanced to express lowering of object-oriented primitives. The original untyped LIR and untyped backend were modified to preserve types. When generating object files, the compiler adds a new section for type information. Other sections in the object file have exactly the same format as the ones generated by the base compiler. The verifier disassembles the code and the type information and then verifies the typed assembly code. A standard linker generates x86 executables from the object files and discards the type information section.

Section 3.1 describes special handling of type variables to allow optimizations. Section 3.2 shows how LILC and SST types are represented in Bartok. Section 3.3 lists the optimizations performed on typed MIR and LIR.

3.1 Type Variables

Type variables are important to guarantee soundness, as shown in Section 2.2. A type variable that identifies the dynamic type of an object should be associated with only that object.

Like traditional typed calculi, LILC introduces a fresh type variable each time an existential type is opened. The type variable identifies the dynamic type of an object. The type variable is in scope until the end of the basic block. The type checker rejects the "Unsafe" function in Section 2.2 (translated to LIR):

$$
\begin{array}{ll}
(1)\ p' = \text{open}\langle\beta\rangle(p); & //\ p' : \beta \\
(2)\ q' = \text{open}\langle\gamma\rangle(q); & //\ q' : \gamma \\
(3)\ vtable = p'.vtable; & \\
(4)\ dist = vtable.distance & //\ dist : (\exists\delta \ll \beta.\ \delta) \rightarrow \text{int} \\
(5)\ dist(q') &
\end{array}
$$

Instructions (1) and (2) introduce distinct type variables β and γ for the dynamic types of p and q respectively. The method "dist" expects an object of β or β's subclasses and q' does not have that type.

This strategy hinders common optimizations, though. For example, two consecutive virtual method invocations on the same object p may be translated to the following LIR code:

$$
\begin{array}{l}
p_1 = \text{open}\langle\alpha\rangle(p); \\
vtable_1 = p_1.vtable; \\
m_1 = vtable_1.m_1 \\
m_1(p_1, \ldots) \\
p_2 = \text{open}\langle\beta\rangle(p); \\
vtable_2 = p_2.vtable; \\
m_2 = vtable_2.m_2 \\
m_2(p_2, \ldots)
\end{array}
$$

The object p does not change between two calls, and thus it is sound to apply common subexpression elimination (CSE) to combine the two opens and vtable fetches. But the two distinct type variables α and β prevent CSE from optimizing the code.

To work around this problem, we separate type variables used in the programs and during type checking, to both allow optimizations and guarantee soundness.

In typed MIR, type variables in programs do not identify dynamic types of objects. It is not required that each open instruction introduces a fresh type variable. In fact, type variables are grouped by their bounds. Two type variables that have the same upper and lower bounds are considered the same. The bounds of type variables have to be accurate because optimizations may query information about members in the type variables, which relies on the bounds.

Opening two objects with the same existential type can reuse a type variable. For example, the two open instructions in the above example can use the same type variable α. The sharing of type variables allows CSE to optimize the code sequence to:

$$
\begin{array}{l}
p_1 = \text{open}\langle\alpha\rangle(p); \\
vtable_1 = p_1.vtable; \\
m_1 = vtable_1.m_1 \\
m_1(p_1, \ldots) \\
m_2 = vtable_1.m_2 \\
m_2(p_1, \ldots)
\end{array}
$$

Note that if p changes between two calls or we call two methods on two distinct objects, CSE cannot unsoundly optimize the code because it cannot tell whether the two opens are the same.

The discrepancy of type variables used in programs and in type checking may cause problems at control merge point. Suppose basic block B1 has an open instruction $x = \text{open}\langle\alpha\rangle(p_1)$ and block B2 has $x = \text{open}\langle\alpha\rangle(p_2)$ and B1 and B2 merge to block B3. This is acceptable in the program because x has only one type α. But during type checking, the checker introduces two type variables for x in B1 and B2, which results in disagreement of x's type in B3.

Typed MIR solves this problem by specifying for each basic block its precondition, including for the type variables visible in the basic block. In the above merge example, block B3 uses a new type variable merged from the ones in B1 and B2 for x's type.

Optimizations may move the code across basic block boundaries. To free the optimizations from maintaining preconditions of basic blocks, typed MIR uses a two-phase type checking. The first phase infers the precondition for each basic block, and the second phase checks the block.

The first type-inference phase mainly deals with merging type variables. Merging is similar to a union operation. Merging a set of unrelated type variables produces a fresh type variable that tracks the components from which the type variable was merged. Merging a type variable with one of its components results in the type variable itself.

For efficiency, the type checker performs a few straightforward optimizations: variables whose types contain no type variables are not tracked in type-checking environments because their types do not change; if a method has only such variables, then type inference is not applied.

3.2 Type Representations

The implementation of LILC and SST requires many new types such as type variables, existential types, and polymorphic types. Naively adding them to the compiler would incur significant changes to the existing code, including type representations, type checking, and optimizations. Our implementation changed or added only 10% of the code in the compiler, because of the choices we made in type representation and type checking.

3.2.1 Typed MIR Representation

Bartok, which is written in C# itself, uses a set of classes to represent MIR types, which include classes in the source programs, function types, and so on. **ClassType** represents classes. **InterfaceType** represents interfaces. **FunctionType** represents function types. These classes form a hierarchy naturally. For example, ClassType and InterfaceType are subclasses of **NamedType**. **IrType** is the root class. Each instance of these classes corresponds to a type.

An instance of ClassType represents a class in the source program. The instance contains all information related to the corresponding class, and has a table of all the members (fields and methods) of the class.

Typed MIR views an instance of ClassType as a combination of both an LILC class name and the corresponding record type that describes the class layout. The instance has both name-based information (class name, superclasses and interfaces) and structure-based information (members). This way, typed MIR can reuse ClassType without change, yet still preserve the two views of classes in LILC. Another benefit of reusing ClassType is that typed MIR does not need to coerce between objects and records, which saves many coercion instructions and makes the code more concise. Those coercions have no runtime overhead, though.

The class name C (an instance of ClassType) in typed MIR also serves as an encoding of the LILC existential type $\exists\alpha \ll C.\ \alpha$. As a result, optimizations can remain as they are: they do not have to deal with the new existential types. Also, the encoding makes it unnecessary to coerce between existential types and class names each time a field is fetched from an object or an object is created. Typed MIR has a new type **ExactClassType** to represent "exact" classes.

In a few cases, typed MIR still needs explicit existential types for those types that don't have the format $\exists\alpha \ll C.\ \alpha$ and therefore cannot use the encoding mechanism. An example is the interface table entry. For this purpose, typed MIR has a class **ExistentialType**.

Typed MIR adds a class **TypeVarClass** for type variables that will be instantiated with classes and interfaces.

To represent the explicit implementations of tags (or runtime types) in the Bartok compiler, typed MIR introduces **RuntimeType** to model tags. An instance of RuntimeType that represents the tag of class C, denoted as "Runtime(C)", contains a reference to the representation of class type C. Tags of different classes have different types.

Similarly to tags, typed MIR introduces a new type **VTableType** for vtables. An instance of VTableType that represents the vtable of class C (denoted as "Vtable(C)") contains a reference to the class C.

The new types ExactClassType, TypeVarClass, RuntimeType, and VTableType all inherit from ClassType. Therefore, each instance of these types is regarded as a normal class and contains a table of its members. The members are added by need to the table. For example, virtual methods can be fetched as normal fields out of vtables.

3.2.2 Typed LIR Representation

The implementation of Typed LIR extends SST to support stack-allocated large values, such as floating-point numbers and structures. Basic SST assumes that stack-allocated values are word-sized.

Typed LIR changes the symbolic representation for stack locations to "$\rho + n$", meaning n bytes between the location labeled by ρ. For aliased locations, the offset n is always 0. Symbolic representations for stack locations in SST are not suitable for typed LIR, because the difference between two stack locations varies. The new location representation makes it easier to tell whether a location is an aliased location and to compute the new location when moving along the stack.

Typed LIR also extends SST to support other features such as objects, classes, and arrays. In fact, typed LIR can reuse most types in typed MIR, for example, class types, interface types, array types, existential types, etc. To avoid duplicating the types, Bartok allows converting an MIR type to an LIR type by simply wrapping the MIR type.

3.3 Optimizations

Our new compiler supports more than 40 optimizations of the base Bartok compiler, with only 2 optimizations unsupported (see Section 3.3.3). Our experience shows that most optimizations can be easily modified to preserve types. This is partly due to our effort to design suitable type representations.

3.3.1 MIR Optimizations

Major optimizations performed at MIR level are as follows:
- Copy propagation
- Constant propagation
- Constant folding
- Common subexpression elimination (including redundant load elimination)
- Dead-code elimination
- Loop-invariant removal
- Reverse copy propagation of temporaries, which transforms "$t = e; v = t$" to "$v = e$"
- Optimizing convert instructions, which compacts chains of convert instructions
- Jump chain elimination, which removes jumping to jump instructions
- Short circuiting, which removes redundant tests of boolean variables
- Loop header cloning, which turns while loops into do-while loops
- Inlining (with heuristics to limit code size increase)
- Elimination of unreachable classes, methods, and fields (treeshaking)

Because our original MIR has type information already, the implementation of typed MIR only needs to change three optimizations. Others are performed on typed MIR as they are on the original MIR. Two optimizations CSE and treeshaking were changed to support the new operators (such as "open") and new types. CSE needs to index all subexpressions. Treeshaking analyzes all instructions to determine which types are accessed. The inlining optimization was changed to support cloning new operators and to support inlining of polymorphic functions. The changes are local and straightforward. The fact that we can reuse almost all optimization code confirms our typed MIR design choices.

3.3.2 LIR Optimizations

Major optimizations on LIR include:
- Copy propagation, including stack locations
- Constant propagation
- Dead-code elimination
- Jump chain elimination
- Reverse common subexpression elimination of load effective address computations
- Peephole optimizations
- Elimination of redundant condition code setting
- Boolean test and branch clean up
- Floating point stack optimizations
- Conversion of "add" to "lea" (load effective address), for example, "ecx = eax + ebx" to "lea ecx, [eax + ebx]"
- Graph-coloring register allocation
- Code layout
- Conversion of "switch" instructions to jumping to entries in jump tables

Changes on LIR optimizations are mostly to propagate types, since the original LIR was untyped. A few optimizations need more significant changes.

The optimization that converts "add" instructions to "lea" instructions may introduce invalid effective addresses. It is still safe because the addresses will never be used to access the memory. But the type checker needs to differentiate such cases and prevent those addresses from being used unsafely.

During register allocation, the compiler fixes the stack frame and assigns stack slots to callee-save registers, function arguments, and spills. We need to record the types of those stack-resident values to type check stack allocation upon entry to a function. The checker needs to know the intended types for the newly-allocated stack slots because of stack-allocated structures. Otherwise, the checker has difficulty finding out the boundaries of slots. The original SST did not need such annotations because each slot was word-sized.

When jump tables are created for translating "switch" instructions, each entry in the jump table can be viewed as a function entry point. We need to give types as preconditions to the entries in the jump table. All entries have the same preconditions, so the jump table can be typed as an array of function pointers.

3.3.3 Unsupported Optimizations

The base Bartok compiler optimizes memory allocation by inlining the allocator implementation directly into the compiler-generated code. Currently the TAL compiler cannot type-check the internal implementation of the memory allocator, so it cannot support this inlining. Our disabling of this optimization is responsible for most of the difference in execution time between code generated by the base Bartok compiler and the type-preserving compiler, as reported in section 6.

Also, as explained in Section 4, we do not support the Array Bounds Check elimination on Demand (ABCD) algorithm.

4. Arrays and Proofs

Bartok implements several optimizations that can eliminate run-time bounds checks from array accesses. First, the common subexpression elimination attempts to consolidate repeated bounds checks for the same array element. For example, the C# expression "a[i] = 1 - a[i]" makes two accesses to the same array element; Bartok's common subexpression elimination removes the bounds check from the second access. Second, an induction variable analysis looks for loops where an array index variable is initialized to a non-negative number, is incremented by one in each loop iteration, and is checked against an array length in the loop condition. Third, Bartok implements the Array Bounds Check elimination on Demand (ABCD) algorithm [1], which infers unnecessary bounds checks by solving a system of difference constraints (constraints of the form $x \leq y + c$, where c is a constant).

This section describes our extensions to Bartok to generate proofs of array access safety, including proofs for run-time bounds checks and proofs for checks eliminated by common subexpression elimination and by induction variable analysis, and then discusses why we were unable to generate proofs for Bartok's ABCD algorithm. We illustrate the proof generation with a simple C# example:

```
public static void Main(string[] args) {
   for (int i = 0; i < args.Length; i++) args[i] = null;
}
```

With our proof-generation extensions, Bartok automatically generates the following TAL code for the Main method's inner loop (omitting the stack types and irrelevant register types, and renaming and reformatting for clarity):

```
B1 ($\alpha$:Int, $\beta$:Int, $\gamma$:Obj, ai:(ArrIndex $\gamma$ $\alpha$),
   al:(ArrLen $\gamma$ $\beta$), gl:(Ge0LtMax $\alpha$))
   {eax:$S_{\text{int32}}(\alpha)$, ebx:$S_{\text{int32}}(\beta)$, ecx:$S_{\text{string[]}}(\gamma)$} =
mov (dword ptr [ecx + eax *4+8] using ai), 0
add eax, (1 : $S_{\text{int32}}(1)$)
; coerce eax to $S_{\text{int32}}$(Succ $\alpha$) using (addToSucc $\alpha$)
cmp eax, ebx
jl(p:Lt (Succ $\alpha$) $\beta$) B1(Succ $\alpha$, $\beta$, $\gamma$,
   incrIndex $\gamma$ $\alpha$ $\beta$ p gl al,
   al, incrGe0LtMax $\gamma$ $\alpha$ $\beta$ p gl al)
```

In this code, the basic block B1 is polymorphic over three type variables: α represents an array index, β represents an array length, and γ represents an array. B1 is also polymorphic over three proof variables. First, ai is a pre-condition that requires ArrIndex γ β, which is defined to be equivalent to $0 \leq \alpha <$ ALen(γ), where $ALen(\gamma)$ is the length of the array γ. Similarly, al and gl require that $\beta =$ ALen(γ) and $0 \leq \alpha <$ MAX, where MAX is the maximum 32-bit signed integer. (We use the optimization-specific abbreviations ArrIndex, ArrLen, and GeLtMax to reduce the generated annotation size and to reduce verification time.) The registers eax, ebx, and ecx hold the index, array length, and array, respectively, where the type $S_{\text{int32}}(X)$ is the singleton type of 32-bit signed integers equal to X and $S_{\tau[]}(X)$ is the singleton type of arrays equal to X.

The first instruction of the loop moves null into array element α, where ecx + 8 is the address of element 0, and each element occupies 4 bytes. The type checker demands a proof that eax hold a valid index for array held in ecx; the annotation "using ai" supplies this proof. (Bartok can also generate proofs for the more complicated address computations that occur for arrays of large C# structs, but we discuss just the simplest case here.) The add instruction increments eax, coercing the resulting singleton type from $\alpha + 1$ to Succ α for conciseness, where Succ stands for successor. The cmp and jl instructions branch back to the beginning of the block if $\alpha + 1$ less than the array length β. The branch requires instantiations of B1's type variables and proof variables with valid types and proofs to satisfy B1's pre-condition. In this case, the axiom incrIndex proves that Succ α is an in-bounds array index, given proofs that Succ $\alpha < \beta$, that $0 \leq \alpha < MAX$ (so that $\alpha + 1$ doesn't overflow), and that $\beta =$ ALen(γ). Here, the jl instruction supplies the proof variable p, asserting that Succ $\alpha < \beta$ in the taken branch.

Broadly speaking, there are two ways a compiler might generate proofs of array access safety. First, a compiler can generate the proper proofs during the array bounds check introduction and during the array bounds check elimination optimizations. In this case, the compiler must preserve the proofs through subsequent compiler phases, using proof-preserving compilation [14]. Alternately, a compiler can postpone proof generation until after all compiler phases have completed, and try to infer proofs by analyzing the compiler-generated assembly language code.

We decided to use some of both approaches, to compare their relative strengths. We postpone proof generation until the conversion from MIR to LIR, which happens well after the HIR's introduction of explicit array bounds check instructions, and after MIR's common subexpression elimination and induction variable optimizations. We change Bartok to annotate HIR and MIR code with hints that highlight the variables and constants used for arrays, array indices, and array lengths. Just before MIR-to-LIR conversion, Bartok runs two dataflow analyses. The first analysis propagates the highlighted variables and constants forward as they flow from one variable to another. The analysis classifies these variables and constants into equivalence classes at each program point, introducing a type variable for each equivalence class. In the example above, α represents the equivalence class containing just the variable eax. The second analysis computes properties about the equivalence classes (for example, β holds the length of array γ); each property becomes a proof variable (e.g. "al"). If the analyses fail to find a safety proof for an array access, they insert an extra run-time bounds check; this ensures that compilation to verifiable TAL can proceed even if the HIR and MIR optimizations outsmart the analyses.

After conversion to LIR with proof annotations, the compiler preserves the proofs through the LIR phases. For example, the LIR jump chain elimination optimization takes jumps from block Bi to block Bj, where Bj consists of a single jump to block Bk, and modifies Bi to jump directly to Bk. Both the Bi-to-Bj jump and Bj-to-Bk jump may have proofs that prove their target's precondition, so we modified the compiler to compose these Bi-to-Bj and Bj-to-Bk proofs together to produce valid Bi-to-Bk proofs. No optimization or transformation required anything more sophisticated than this to preserve proofs. Nevertheless, due to the large number of optimizations, we did not implement proof preservation for every instance of every optimization. In particular, constant propagation may want to replace a variable of type $S(\alpha)$ with a constant c of type $S(c)$; rather than fixing the types (e.g. by proving $\alpha = c$), we simply don't propagate c to the $S(\alpha)$ variable in this case.

Our experience so far shows that practical proof preservation is possible, but requires effort proportional to the number of compiler optimizations and transformations. Implementing the proof inference for the MIR-to-LIR conversion required less effort than implementing the LIR proof preservation. Nevertheless, proof preservation has one potential advantage over inference: if the initial input to the compiler already has proofs of array access safety (or other integer-related properties), a proof-preserving compiler can preserve these proofs, even if the proofs are too sophisticated for inference to re-discover [14]. For example, we've used Bartok to compile a hand-written LIR program, hand-annotated with proofs showing that the program correctly computes the factorial, to TAL code annotated with proofs showing correct factorial computation. This works even though Bartok knows nothing about factorials.

4.1 ABCD

We also attempted to generate safety proofs for Bartok's ABCD array-bounds check elimination optimization, but the "proofs" seemed to require axioms that are unsound for modular arithmetic. This observation led to a counterexample:

```
void F(int[] arr, int i) {
  if (i <= arr.Length) {
    int j = i - 1;
    if (j >= 0) arr[j]++;  } }
```

At the "arr[j]++" statement, Bartok's ABCD analysis correctly concludes that $0 \leq j$ and $i \leq$ arr.Length and $j \leq i - 1$, but then incorrectly eliminates the bounds check based on the erroneous conclusion that $0 \leq j \leq arr.Length - 1$. This conclusion fails when i is the minimum signed 32-bit integer, so that i - 1 underflows. The out-of-bounds array store clobbers nearby memory, undermining type safety.

Since the original paper on ABCD [1] did not claim to work for modular arithmetic, it was no surprise that the algorithm might fail for 32-bit integer indices, and in fact Bartok already had heuristics to squelch the ABCD algorithm whenever it saw large integer constants. Unfortunately, the method above contains no constants larger than 1, so Bartok still (incorrectly) eliminates the bounds check. We were disappointed at not generating safety proofs for ABCD, but we were grateful that our attempt led us to discover a real vulnerability in Bartok.

5. GC Information Verification

Object files generated by the compiler also include information used by the garbage collector. This information allows the garbage collector to identify all references on the heap and stack at run time. If this information is incorrect, the garbage collector may attempt to trace non-reference data or prematurely reclaim live data, both of which can lead to safety violations. Thus, in order to ensure safety, we must check that all of the garbage collection information in the object files is consistent with our type information.

The compiler generates three kinds of information for the garbage collector's use:

The *static data table* is a bitmap that indicates which words in the static data section contain traceable references.

The *object table* in each class's vtable contains information that allows the garbage collector to locate reference-containing fields in instances of that class.

The *stack table* maps every valid return address in the program to information about how to find references in the caller's stack frame. At collection time, the collector walks the stack, using the information for each return address on the stack to find references in each frame.

For each kind of information, we must verify that the garbage collector's information is consistent with our type information. Checking the static data table is straightforward: we simply compare each bit in the bitmap with the type of the corresponding word of memory. Checking the object tables is similar: each vtable object in static data is identified by a special type, so we can extract the word containing pointer information and use it to check the corresponding class's type information. We give this word a singleton type that depends on the class, which ensures that it cannot be altered and that invalid vtables cannot be constructed by user code. We currently omit checks for fields of classes defined in other compilation units when those fields are not directly accessed by the user's code.

Checking stack tables is the most involved aspect of our garbage collector verification. After checking each call instruction, we look up the garbage collection information for the associated return address. This information indicates which arguments, spill slots, and callee-saved registers contain references, which can be compared against the stack type at that call instruction. In addition, this information indicates whether callee-saved registers have been saved on this stack. In order to ensure that these values are traced by the collector, we must ensure that they were correctly saved to the stack, which we can do by comparing the fresh type variables assigned to each callee-saved register with the type of the corresponding stack slot. Finally, we must ensure that the return address saved on the stack has not been replaced with a return address that happens to have the same type but is not a valid index into the garbage collector's tables. To do so, we refine the function pointer type to create an immutable function pointer type that can only be pushed on the stack with a "call" instruction and can only be popped off the stack with a "ret" instruction.

Finally, we must ensure that any write barriers required by a concurrent garbage collector have been placed appropriately by the compiler. In our compiler, write barriers are implemented as a method invocation that does any appropriate checks in addition to performing the write. Thus, we can check write barriers by making locations that require barriers immutable; any writes to these locations must be performed by invoking the write barrier, which is trusted code outside of the object file. Currently, we use a collector that does not require write barriers.

When implementing these checks, we found it helpful to adapt the source code for the runtime system directly for use in the verifier. The encodings for these garbage collection tables are quite complex, so we reused the runtime system's table decoder, adding further checks where necessary to guard against invalid encodings. To design the checker, we abstracted the garbage collector's reference scanning code, replacing all direct memory references with references to the associated type information. This approach gave us greater confidence that our verifier's checks match the assumptions of the runtime system.

6. Measurements

The compiler has about 150 benchmarks for day-to-day testing. This section shows the numbers for the seven largest ones (Table 1).

The performance numbers were measured on a PC running Windows Vista with two 2.66GHz CPUs and 4GB of memory. We use separate compilation mode to compile the benchmarks separately from the libraries. The runtime uses a mark-sweep garbage collector. The running times are averaged over five runs of each program.

Execution Time. Table 2 shows the performance comparison between TAL, the base-compiler-generated code, and CLR-generated code. The fourth column "TAL/Base" is the second column "TAL" divided by the third column "Base". The TAL code is 0.95-1.09 times slower than the base compiler's generated code, with a geometric mean of 1.02. The TAL code is slightly less efficient than the base compiler-generated code because the new compiler does not support inlining of heap allocations. The two benchmarks ahcbench and asmlc are allocation-intensive, and therefore are affected most. The TAL code is nearly as efficient as the base-compiler-generated code if the base compiler turns off inlining of heap allocation.

Name	Description	**Executable Size** (in bytes)
ahcbench	An implementation of Adaptive Huffman Compression.	4,820,992
asmlc	A compiler for Abstract State Machine Language.	20,586,496
lcscbench	The front end of a C# compiler.	7,766,016
mandelform	An implementation of mandelbrot set computation.	12,775,424
sat_solver	An implementation of SAT solver written in C#.	4,943,872
selfhost1421	A version of the Bartok Compiler.	7,847,936
zinger	A model checker to explore the state of the zing model.	13,074,432

Table 1. Benchmarks.

Benchmark	TAL	Base	TAL/Base	CLR
mandelform	1262.93	1234.92	1.02	1423.14
selfhost1421	256.30	235.46	1.09	93.99
ahcbench	4.08	3.75	1.09	4.50
lcscsbench	6.82	6.66	1.02	10.55
asmlc	0.96	0.92	1.04	1.58
zinger	2.89	3.01	0.96	3.32
sat_solver	4.50	4.76	0.95	4.31
geomean			1.02	

Table 2. Execution Time (in seconds)

Benchmark	TAL	Base	TAL/Base
mandelform	117088	65445	1.79
selfhost1421	18871862	9012642	2.09
ahcbench	148880	61077	2.44
lcscsbench	14205932	8004809	1.78
asmlc	31851094	18856426	1.69
zinger	2400600	1076470	2.23
sat_solver	883589	351134	2.52
geomean			2.05

Table 4. Object File Size (in bytes)

Benchmark	TAL	Base	TAL/Base
mandelform	16.15	15.65	1.03
selfhost1421	1087.99	63.54	17.12
ahcbench	6.37	6.24	1.02
lcscsbench	153.60	78.09	1.97
asmlc	158.78	160.76	0.99
zinger	25.59	17.68	1.45
sat_solver	9.64	7.18	1.34
geomean			1.83

Table 3. Compilation Time (in seconds)

Benchmark	Checking Time (% of Compilation Time)
mandelform	1.61 %
selfhost1421	5.67 %
ahcbench	4.40 %
lcscsbench	18.78 %
asmlc	2.14 %
zinger	18.13 %
sat_solver	5.87 %
geomean	5.59 %

Table 5. Checking Time Compared with Compilation Time

Compilation Time. Table 3 shows the performance comparison of the TAL compiler and the base compiler. The TAL compiler is 0.99-17.12 times slower than the base compiler, with a geometric mean of 1.83. The slowdown is largely due to type inference and writing type information to object files. Typed MIR and LIR have type inference to infer the precondition of each basic block. Typed LIR type inference and writing type information to object files account for 49% of the compilation slowdown in lcscbench, 20% in zinger, and 13% in sat_solver, and 70% in selfhost1421. The benchmark selfhost1421 has significant slowdown because of inefficient type inference for arrays. Other slowdown is due to a larger number of IR instructions and MIR/LIR types. Typed MIR and LIR insert coercion instructions to change types, most of which are no-ops at runtime.

Object File Size. Table 4 shows the size comparison of object files generated by the TAL compiler and the base compiler. The TAL object files are significantly larger than those generated by the base compiler: 1.69-2.52 times larger, with a geometric mean of 2.05. We decided to make the TAL checker simple because the checker is in the TCB. Therefore, the TAL programs are extensively annotated with types. Excluding the type information, the object files generated by the TAL compiler are as small as those generated by the base compiler. A smarter checker might not require as much type information because it could infer type information, but would also be more difficult to trust or verify than a simple checker.

Type Checking Time. Table 5 shows the TAL checking time compared with compilation time. The TAL checking time is 1.61%-18.78% of the compilation time, with a geometric mean of 5.59%. Type checking TAL is fast, as in prior TAL compilers.

7. Related Work

Many compilers use typed intermediate representations. We compare our compiler with a few ones that preserve types to assembly code. None of them has as many optimizations as Bartok has, or supports compiler-runtime interface verification.

SpecialJ [4] is a certifying compiler for Java. SpecialJ produces PCC binaries (code with a safety proof), which provides a more general framework than TAL. It supports exception handling via pushing and popping handlers. (Our system uses exception handling tables, and we cannot yet check the tables.) SpecialJ treats all runtime libraries as primitives, and thus it cannot verify them or inline them into user programs. The SpecialJ paper used hand-optimized code as examples and mentioned only two optimizations that SpecialJ supported.

The TALx86 compiler is a type-preserving compiler from a C-like language to x86 code [10], but it does not perform aggressive optimizations such as array bound check elimination. The type system lacks support for many modern language features such as CIL's by-reference parameters.

The LTAL compiler compiles core ML programs to typed SPARC code [3]. It has a minimal runtime (no GC) and does not support proofs about integers.

Leroy implemented a certified compiler from a C-like language to PowerPC assembly code [8], verifying most of the compiler code itself, not just the compiler's output. The compiler has about 5,000 lines of code with few optimizations. The proof of the compiler is about 8 times bigger in size than the compiler. It is difficult to scale to large-scale compilers like Bartok.

Menon *et al.* proposed an SSA presentation that uses proof variables to encode safety information (for example array bounds) [9]. It supports optimizations by treating proof variables as normal variables. SSA is not suitable for TAL. Section 2.4 has discussed other work about verifying array-bounds checking.

Vanderwaart and Crary developed a type theory for the compiler-GC interface, but did not have an implementation [15].

8. Conclusion

In this paper, we have presented a large-scale, type-preserving, optimizing compiler. This work shows that preserving type information at the assembly language level is practical even in the presence of aggressive low-level optimizations. We were also able to implement a lightweight verifier that checks the safety of our compiler's output, including many aspects of its interaction with the runtime system. We believe that type-preserving compilation is a useful approach to minimizing the trusted computing base.

Acknowledgments

We would like to thank David Tarditi, Jim Larus, and Galen Hunt for their guidance and support of this work, David Tarditi and the Bartok team for their help with Bartok-related issues, and anonymous reviewers for their comments on an earlier version of the paper.

References

[1] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array-bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[2] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages*, pages 38–49, 2005.

[3] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[4] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[5] K. Crary and J. C. Vanderwaart. An expressive, scalable type theory for certified code. In *ACM SIGPLAN International Conference on Functional Programming*, pages 191–205, 2002.

[6] Microsoft Corp. et al. *Common Language Infrastructure*. 2002. http://msdn.microsoft.com/net/ecma/.

[7] D. Grossman and J. G. Morrisett. Scalable certification for typed assembly language. In *International Workshop on Types in Compilation*, pages 117–146, 2001.

[8] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[9] V. S. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A. Adl-Tabatabai, and L. Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *ACM Symposium on Principles of Programming Languages*, pages 397–408, 2006.

[10] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[11] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *ACM Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[12] G. Necula. Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[13] F. Perry, C. Hawblitzel, and J. Chen. Simple and flexible stack types. In *International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, July 2007.

[14] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *ACM Symposium on Principles of Programming Languages*, 2002.

[15] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *ACM SIGPLAN workshop on Types in languages design and implementation*, pages 109–122, 2003.

[16] H. Xi and R. Harper. A dependently typed assembly language. In *2001 ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, September 2001.