Efficient Type Representation in TAL

Juan Chen
Microsoft Research
juanchen@microsoft.com

Abstract

Certifying compilers generate proofs for low-level code that guarantee safety properties of the code. Type information is an essential part of safety proofs. But the size of type information remains a concern for certifying compilers in practice. This paper demonstrates type representation techniques in a large-scale compiler that achieves both concise type information and efficient type checking. In our 200,000-line certifying compiler, the size of type information is about 36% of the size of *pure code and data* for our benchmarks, the best result to the best of our knowledge. The type checking time is about 2% of the compilation time.

1 Introduction

Proof-Carrying Code [8] and Typed Assembly Language (TAL) [7] use certifying compilers to generate proofs for low-level code that guarantee safety properties of the code. Type information is an essential part of safety proofs and is used to describe loop invariants, preconditions of functions, etc. The size of type information, though, remains one of the main concerns for certifying compilers in practice. With naive representations, the size of type information can explode. Even with techniques that greatly improve sharing of type representations, such as sharing common subterms among types, type information may still be much larger than code. This is mainly because type checking needs elaborate type information to describe machine states, including the register bank, the heap, and the stack. Large type size is undesirable in practice, especially for large programs.

This paper demonstrates type representation techniques in a large-scale compiler that achieves both concise type information and efficient type checking. In our 200,000-line object-oriented certifying compiler, the size of type information is about 36% of the size of *pure code and data* for our benchmarks, the best result to the best of our knowledge. The type checking time is about 2% of the compilation time. Note here that we compare type information size against code and data size, *not* object file size. Object files contain relocation and symbol information and may be much larger than pure code and data.

Our compiler uses simple yet effective type representation techniques—some shared with existing certifying compilers (mostly for functional languages) and others specialized for object-oriented languages: (1) it shares common subterms of types (CSE on types); (2) it uses function types to encode more elaborate code pointer types; (3) it removes redundant or unnecessary metadata (information about classes and interfaces, including fields, methods, and inheritance); (4) it removes type information for components of large data in the data sections of object files; (5) it uses concise integer encodings.

The contribution of this paper is that it shows that large-scale certifying compilers can have both concise type representation and efficient type checking for TAL. The paper also shows that type representation in object-oriented certifying compilers needs special care with metadata and data.

2 Background

We first give an overview of the compiler, the type checker, and the type system of the target language.

The Compiler. Our certifying compiler (called Bartok) compiles Microsoft Common Intermediate Language (CIL) to standalone typed x86 executables [2]. It has about 200,000 lines of code (mostly in C#) and more than 40 optimizations. Bartok is about an order of magnitude larger than previously published certifying compilers. It is used by about 30 developers on a daily basis. Performance of Bartok's generated code is comparable to that 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.

Bartok compiles CIL through several typed intermediate languages and generates typed x86. Unlike most existing certifying compilers which mingle type information and code, Bartok separates them. It writes type information in a special section in the object file. The sections for code and data use the standard COFF format without any change. A standard linker links the object file with libraries and generates normal x86 executables, the same way it links untyped x86 object files. The linker discards the type information section.

Separating types from code allows us to keep the standard code format, but it needs to record additional information to connect the type with the code, for example, where a type is used. We chose this approach so that Bartok-compiled code can link with the libraries, which are not compiled by the certifying compiler at the moment.

The Type Checker. We would like to make the type checker simple and efficient because the type checker is in the trusted computing base. One design choice is that the type checker checks each basic block in the program only once. The TALx86 compiler [6]—a certifying compiler from a C-like language to x86—has a threshold that decides how many times a block can be checked (4 is the threshold in [5]).

The checker checks one function at a time. A function consists of a sequence of basic blocks. The checker performs a breath-first scan of the control flow graph, starting from the root block.

Checking a basic block requires the block's precondition, i.e., the state of the register bank and the control stack at the beginning of the block. The precondition either comes from type annotations for the block or can be computed by the type checker.

Given the precondition, the type checker checks the instructions in a basic block. Checking an instruction may need additional type information. For example, if an instruction refers to static data, the type checker needs type information for the data. The additional type information for instructions comes from annotations in the type section.

The type checker uses the postcondition of a basic block to compute the preconditions of the successors, if the successors do not have their preconditions already. A block at merge points must have an annotation for its precondition, unless all its predecessors have the same postcondition. This approach, together with the breadth-first scan, guarantees that the checker checks each block only once.

Exception header blocks (for exception handlers) always have annotations for their preconditions because there is no explicit control flow from the root block to the exception header blocks.

The Target Type System. The type system of the target language uses main ideas of LIL_C [3] to represent classes and objects. LIL_C is a low-level typed intermediate language for compiling object-oriented languages with classes. It is lower-level than bytecode and CIL because it describes implementation of virtual method invocation and type cast instead of treating them as primitives.

LIL_C differs from prior class and object encodings in that it preserves object-oriented notions such as class names and name-based subclassing, whereas prior encodings compiled these notions away.

LIL_C uses an "exact" notion of classes. A class name in LIL_C represents only objects of exactly that class, not including objects of subclasses. Each class C has a corresponding record type $R(C)$ that describes the object layout of C, including the vtable and the fields.

Objects can be coerced to or from records with appropriate record types, without runtime overhead. To create an object, we create a record and coerce it to an object. To fetch a field or invoke a method, we coerce the object to a record and then fetch the field or the method pointer from the record.

To represent an object of C or C's subclasses, LIL_C uses an existential type $\exists\alpha \ll C.\ \alpha$, read as "there exists α where α is a subclass of C, and the object has type α". The type variable α represents the object's runtime type and the notation $\ll$ means subclassing. The type variable α has a subclassing bound C, which means that the runtime type of the object is a subclass of C. The subclassing bounds can only be class names or type variables that will be instantiated with class names. Subclassing-bounded quantification and the separation between name-based subclassing and structure-based subtyping make the type system decidable and expressive.

A source class name C is then translated to an LIL_C type $\exists\alpha \ll C.\ \alpha$. If C is a subclass of class B, then $\exists\alpha \ll C.\ \alpha$ is a subtype of $\exists\alpha \ll B.\ \alpha$, which expresses inheritance—objects of C or C's subclasses can be used as objects of B or B's subclasses.

To represent the precondition of a basic block, the target language uses a code pointer type to describe the types of registers and stack locations at the beginning of the block. The code pointer type is elaborate: it specifies the type for each live register and for each stack location in the current stack frame, including the return address.

3 Type Representation

Now we explain the main type representation techniques in Bartok. The first technique shares common subterms among types, a common practice. Traditional sharing is important but not enough. Our compiler combines it with several other techniques: the second technique optimizes representation of code pointer types; the third one compresses representation of metadata information; the fourth reduces type information for data; and the last one improves integer encoding. The third and the fourth ones are specific to object-oriented languages, reducing type size by one third in our measured benchmarks.

One may think that we make the type checker simple but the type decoder complicated because of these type representation techniques. We consider this a good tradeoff because the type decoder is not in the trusted computing base. The result of the type decoder is checked and thus not trusted, just like untrusted type annotations. The type checking fails if the type decoder generates type information incompatible with the code.

3.1 Sharing Common Subterms

Sharing common subterms of types is a well known technique many compilers use ([5, 12]). It is similar to the common sub-expression elimination (CSE) optimization for terms, where two types that have a common subterm share one copy of the subterm, instead of duplicating it in both types.

When writing type information, Bartok first collects types that should be written into the object files. It goes through the whole program, including data, external labels, functions, basic blocks, and instructions, and collects each type it encounters, and records the type in an array. The compiler keeps track of which type it has seen. If it sees a type that has appeared already, nothing needs to be done. Otherwise, it marks the type as seen and continues to collect the components (subterms) of the types. Collecting component types makes it possible to achieve finer-grained sharing between types.

The compiler also tracks where in the program these types are used—the relative location in the code section of the corresponding basic blocks, instructions, etc. Such information is also written into the object file to connect types with the code those types annotate.

Bartok then writes the collected types into the object file. It writes the type array first, which contains all types the program uses. Each type starts with a byte indicating whether the type is a primitive type (e.g. int32) or a type constructor (e.g. array). If the first byte is a type constructor, the following bytes are encodings of the components. Interpretation of the following types depends on the type constructor. For example, if the type constructor indicates this is an array type, the following bytes encode the rank and the element type.

The encoding of a component type is its index in the type array. The index is valid because all component types are collected in the type array as well. The indirection achieves sharing in the object file: two occurrences of the same type share the same encoding.

After the compiler finishes writing the type array, it writes the types for block preconditions, instructions, data, etc. All types are again encoded as their indices in the type array.

When the type decoder reads back from object files the type information, it first decodes the type array and rebuilds the types. Because of the type index representation, the decoded types are hash-consed automatically. This also makes the type checker efficient because type equality tests are simply reference equality tests. Two structurally equivalent types share the same reference.

3.2 Code Pointer Types

The target language uses code pointer types to represent preconditions. As Grossman and Morrisett pointed out in [5], the size of preconditions is the most important factor of type size. Code pointer types are pervasive: they can take up to 20% of types used by programs. Furthermore, code pointer types are more complex and larger than other types because they record machine states. Optimizing code pointer type representation can have significant impact on type size.

There are two uses of code pointer types in Bartok: for method types and for preconditions of basic blocks. The source language and the high-level intermediate representations use function types for methods. The target language uses code pointer types to express explicit control stacks. Function types specify the types for the parameters and the return value and have no reference to the control stack. To check an x86 call instruction, the type checker needs explicit description of the current stack to check if the stack satisfies the requirement of the callee.

One observation is that if we know the function type and the calling convention of a method, we can compute the corresponding code pointer type. Therefore we can use the function type and the calling convention as a more concise representation of the code pointer type, which saves space dramatically. There are only a few calling conventions Bartok uses, so encoding calling conventions is easy. This technique reduces the number of code pointer types by 90% and the total number of types by 76% in our benchmarks.

The type decoder transforms function types to code pointer types according to their calling conventions. The type checker does not see any function types at all, therefore does not need any special handling for function types.

This approach has the drawback that the decoder has to hard-wire the calling conventions the compiler uses. But note here that the compiler still has the flexibility to use more than one calling convention or choose from various calling conventions. We require only that the decoder be aware of all the calling conventions the compile may use. We think the benefits of using function types outweigh the drawback.

We have addressed the code pointer types for methods, but preconditions for basic blocks still need code pointer types and they may not have corresponding function types. Bartok reduces the number of preconditions by omitting preconditions of basic blocks with only one predecessor or with multiple predecessors where the post conditions of all predecessors agree. Such preconditions can be reconstructed easily.

For the remaining code pointer types, Bartok optimizes common patterns, for example, sharing types of callee-save registers in preconditions. Bartok uses the standard approach to handle callee-save registers: at the entry point of a method, each callee-save register is assigned a type variable. The return address requires that the callee-save register have the same type variable to guarantee that the value of the register has been restored when the method returns. Every method has type variables for these callee-save registers. Those type variables are carried into preconditions of each basic block. Furthermore, if a callee-save register is not assigned a new value, its type remains the original type variable. The compiler shares the type variables for those registers and uses bits in preconditions to indicate if a callee-save register still has not changed its value.

3.3 Class Metadata

Object-oriented programming languages use metadata extensively to represent essential information about classes and interfaces, such as fields, methods, and inheritance. Object-oriented programs are organized by classes. Classes may contain many fields and methods and thus require large metadata. Also it is common for object-oriented programs to refer to classes defined in libraries or in other compilation units. Metadata for those externally defined classes is needed as well.

Checking core object-oriented features such as field access and virtual method call relies on metadata. To type check an instruction that fetches a field from an object, the checker needs to know the type of the field. Similarly, to type check a virtual method call instruction, the checker needs to know the type of the method in order to check whether the arguments have the desired types. The information about field types and method types is represented as metadata for the class.

At assembly language level, all field accesses and method accesses are lowered to memory accesses, with field/method names lowered to integer offsets from the beginning of the object reference/the vtable. Metadata in the target language records mapping from offsets of fields and methods to their types.

Metadata in the target language also includes inheritance information, for example, the superclass and superinterfaces of a class. This is important for the type checker to decide if a class/interface is a subclass of another class/interface.

Bartok includes in the metadata of a class the following information: field offsets and types, virtual method offsets (in vtables) and types, the superclass, and the superinterfaces. For interfaces, metadata contains superinterfaces and interface method offsets (in interface tables) and types.

Bartok has the following techniques to reduce metadata information in object files. First, it shares metadata information between superclasses and subclasses. A subclass contains all fields and methods the superclass has. It is unnecessary to duplicate the information for those fields and methods in the subclass metadata.

Second, for external classes defined in other compilation units, the compiler records only information of the fields and methods referenced in this compilation unit. The compilation unit that defines the external class writes full metadata for the class. Checking the consistency of metadata in different compilation units is considered future work because currently the linker discards type information in object files.

Third, Bartok excludes in the metadata useless private fields–those not referenced in the compilation unit where the enclosing class is defined.

3.4 Data

Similar to metadata, object-oriented programs use more data than functional or imperative ones. For each class, the compiler creates a vtable that contains all virtual methods of the class, including the ones from its parent class. Also, the compiler creates a runtime tag for the class, which uniquely identifies the

class at run time. Runtime tags are used for type casts. Vtables and runtime tags are put into the data section of object files.

The checker needs to check the data to see if they match their specified types. For example, we cannot claim an integer in the data section to have a pointer type.

Vtables and runtime tags are complicated data structures. A vtable in Bartok contains a fixed section (84 bytes) and a list of virtual method pointers. The fixed section is the same for all vtables, including a pointer to the runtime tag of the corresponding class, a pointer to the interface method table, an array of pointers to runtime tags of superclasses, a pointer to the runtime tag of the element type if the corresponding class is an array, and many other fields. A runtime tag in Bartok needs 80 bytes, including a pointer to the corresponding vtable, a pointer to the runtime tag of the parent class, interface table information, etc.

The structure of those data is flattened in the object files in the sense that a record of two integers looks exactly the same as two independent integers. When writing a vtable into the data section of object files, Bartok writes first a label indicating the beginning of the vtable, then each piece of information in the fixed section, and at last a list of virtual method pointers. Each part of the vtable is represented as a piece of data, independent of each other. The only requirement is that they have to be adjacent and in order. Runtime tags have similar encodings.

One approach to represent the types for vtables and runtime tags is to record the type for each piece of data in the data structures. Using this elaborate representation for data allows the compiler to choose vtable layout and runtime tag layout. The compiler even has the flexibility to use different layouts for vtables and runtime tags of different classes. But the type information for these large data structures takes significant space.

Instead, Bartok bakes in the layout strategies of vtables and runtime tags. It records only the type of the starting label as the vtable or the runtime tag of the corresponding class. The type information for the following components can be inferred from the layout strategies of vtables and runtime tags and the metadata for the corresponding class. Often an object-oriented compiler uses a uniform layout strategy for all vtables and runtime tags, so this is not a severe restriction for the compiler.

3.5 Integer Encoding

Bartok uses many integers in its type representation. The table-based type sharing encodes component types as integer indices into the type array. Field and method offsets in metadata are integers. Type information for basic blocks and instructions needs to track the locations (offsets from the beginning of the code section) of the basic blocks and the instructions.

Making integer representations concise has a surprisingly big impact on the whole size of type information. Bartok first used a naive four-byte representation for integers, and then changed to a more space-efficient one, reducing the total type size from 64% of code and data size to 36%.

The compressed integer representation is adopted from the one used by CIL for signatures, with slight changes to represent large integers. Integers between 0 and 0x7F are encoded as one-byte integers as they are (the highest bit—bit 7—is 0). Those between 0x80 and 0x3FFF are encoded as two-byte integers with the highest two bits (bits 15 and 14) 10. Integers between 0x4000 and 0x1FFFFFFF are encoded as four-byte integers with the highest three bits (bits 31-29) 110. Other integers are represented as five-byte integers with the first byte 0xFF and the rest four bytes for the integer value. Many integers are represented with fewer than 4 bytes because many type indices and offsets fall into the range between 0 and 0x3FFF.

4 Measurement

This section explains the experimental results and effect of the type representation techniques. Bartok has about 150 benchmarks for day-to-day testing. The table below shows seven large ones and the size of their code and data. The other parts in object files, including relocation information and symbol information, are not counted. Three benchmarks —asmlc, selfhost1421, and lcscbench—are large, code and data size ranging from 2.5MB to 5.4MB. The other four are small, code and data size ranging from 7.7KB to 271KB.

Name	**Description**	**Size of pure code and data** (bytes)
ahcbench	An implementation of Adaptive Huffman Compression.	13,908
asmlc	A compiler for Abstract State Machine Language.	5,436,519
selfhost1421	A version of the Bartok Compiler.	2,507,286
lcscbench	The front end of a C# compiler.	2,944,867
mandelform	An implementation of mandelbrot set computation.	7,702
sat_solver	An implementation of SAT solver written in C#.	135,571
zinger	A model checker to explore the state of the zing model.	271,368

The measurement uses the separate compilation mode of Bartok—user programs are compiled separately from the libraries. The type checker checks the object code compiled from the user programs. We expect that less type information is needed to type check executables than checking object code because metadata for cross reference between compilation units is no longer necessary. Currently if a unit A refers to a class defined in another unit B, the object code for unit A must contain enough metadata for the class. Executables do not need such metadata. The linker just needs to check if the two units have consistent metadata.

4.1 Effect of Type Representation Techniques

With sharing common subterms but no other techniques, the type information can be several times larger than the code and data. Applying our type representation techniques reduces type size from about 2.3 times of code and data size to 36% (geometric means).

Figure 1 summarizes the effect of type representation techniques for the benchmarks. The X axis is the benchmarks. The Y axis is the type size normalized by the code and data size (the code and data size is 1. The lower bars, the better). For each benchmark, the left most bar is the code and data size. The next bar is the type size with only sharing common subterms. Each of the other bars shows the type size with one more technique applied (improve upon its adjacent left bar), till the rightmost bar which shows the type size with all techniques applied. The explanation of individual techniques is as follows.

Code Pointer Type. As explained earlier, code pointer types have the biggest impact on type size. It pays to optimize the representation of code pointer types as much as possible. Simply using function types to encode code pointer types reduces type size from 230% of code and data size to 101%.

Eliding Block Preconditions. Besides using function types, Bartok reduces the number of code pointer types by eliding preconditions of some basic blocks. This technique has a much smaller impact (type size reduced from 102% to 96%) than the previous one although it significantly reduces the number of blocks that need preconditions. The preconditions of about 60% of the basic blocks are removed. The reason for the small impact is that basic blocks in a function tend to share many subterms, such as the type of the return address and the stack (if the stack is the same at the beginning of those blocks). Most merge points still need code pointer types for their preconditions. The subterms are encoded even if many blocks' preconditions that share the subterms are removed.

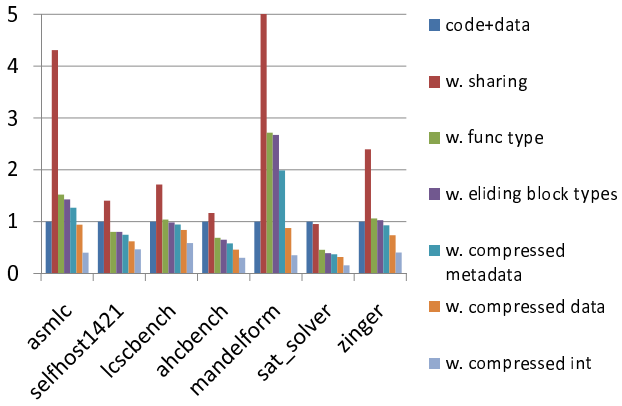

Figure 1: Effect of Type Representation Techniques

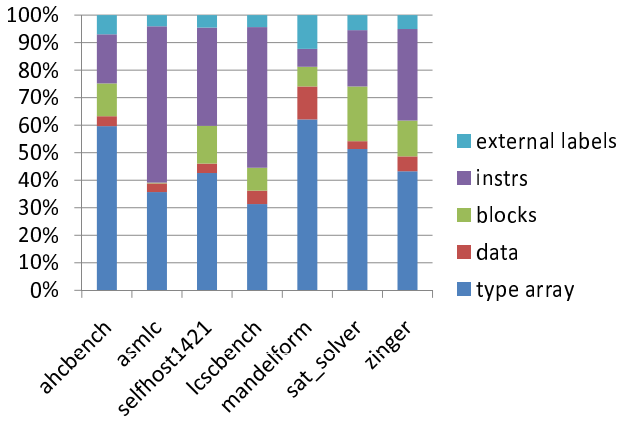

Figure 2: Type Information Breakdown

Compressed Metadata. With efficient representation of code pointer types, we further compress metadata by sharing metadata between superclasses and subclasses and removing unnecessary field or method information. This technique reduces type size from 96% of code and data size to 86%.

Compressed Data. Type size is further reduced to 64% of code and data size after we remove types for small data pieces in vtables and runtime tags. This technique reduces significantly the number of data that need types. Only 14% data need to encode their types after this technique. A large portion of the data section is for vtables and runtime tags.

Compressed Integers. Integer compression reduces the type size from 64% of code and data size to 36%. The reduction is because of the extensive usage of integers in type encoding, such as type indices and locations.

4.2 Type Information Breakdown

Now with all of the discussed techniques applied, let us look into what is in the type section of object files. Figure 2 shows a summary (all in percentage of the whole type size). Note that this shows just an estimate of relative sizes of various type information. The type array contains types for all data, blocks, and instructions. It is not surprising that the type array takes the largest chunk (45% of the type section). Instruction type is the second largest (20%). Type coercions take a significant portion of the instruction types. Types for data take only 4%, the smallest, due to the data compression technique.

4.3 Checking Time

We measure the type checking time to show the efficiency of the type checker. The performance numbers were measured on a PC running Windows XP SP3 with two 3.6GHz CPUs and 2GB of memory. The table below lists the absolute checking time in seconds and the checking time relative to the compilation time. Checking is fast and takes only about 2% of the compilation time. The checking time is roughly proportional to the code and data size, with the exception of the smallest benchmarks (ahcbench and mandelform), which is likely because of measurement accuracy.

Benchmark	**Checking Time**(sec)	**% of Comp Time**
ahcbench	0.11	1.2 %
asmlc	11.4	3.2 %
selfhost1421	4.90	3.6 %
lcscbench	3.13	2.6 %
mandelform	0.08	0.3 %
sat_solver	0.22	1.7 %
zinger	0.59	1.8 %
geomean		1.7 %

5 Related Work

The most relevant work to this paper is the TALx86 compiler, where type information was reduced to about 50% of the object code [6, 5]. The type representation of Bartok differs from that of TALx86 in the following aspects. First, Bartok's source language is object-oriented, thus Bartok has to encode type information for a large amount of data and metadata required by object-oriented programming, such as vtables. Techniques more relevant to object-oriented languages, such as compression of metadata and data, prove to be important for Bartok. Second, Bartok differs from TALx86 in several designing choices. It separates type information from code and requires that each basic block be checked only once. The TALx86 type checker may check a basic block multiple times (for different paths). For encoding of code pointer types, Bartok uses function types whereas TALx86 used parameterized abbreviation (type-level functions to abstract common structures of the code pointer types). Parameterized abbreviation may achieve more concise representation of code pointer types. It is considered future work to adopt this technique in Bartok.

Necula *et al.* developed a certifying compiler called SpecialJ for Java [4] as part of the Proof-Carrying Code framework [8]. The compiler translates Java bytecode to x86 assembly code with type annotations, and a safety proof that certifies the code. Later work greatly reduced the size of safety proof [10]: instead of encoding the complete proof, the compiler encoded only hints (oracles) to the proof checker so that the checker can determine which rule to apply when it has multiple choices. Type information and metadata are not included in the measurement of safety proof size.

Necula and Lee also developed a logical framework LF_i that allows reconstructing proof subterms during proof checking, so that those subterms do not need to be encoded in the proof [9]. This results in both compact representation of proofs and efficient proof checking.

Some compression techniques for Java class files reorganized the files, including data, metadata, and code, to achieve better sharing and compression [11, 1]. Many of the techniques, e.g. reference compression, may be applied to our type representation to further reduce type size.

Shao showed a few effective techniques in the FLINT compiler that reduce the memory consumption of types [12]. One of the techniques is hash-consing. The source language of the FLINT compiler is SML, a functional language. There is not much metadata information the compiler needs to track. Also,

the FLINT intermediate language does not need to handle the control stack or code pointer types because the compiler has not exposed the stack yet at this stage.

6 Conclusion

This paper explains how our large-scale object-oriented certifying compiler achieves both concise type representation and efficient type checking for TAL. In our experience, efficient encoding of information specific to object-oriented languages—metadata and data—has a big impact on type information size.

References

[1] Q. Bradley, R. Horspool, and J. Vitek. Jazz: An efficient compressed format for java archive files. In *Proceedings of CASCON '98*, 1998.

[2] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 183–192, New York, NY, USA, 2008. ACM.

[3] 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.

[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] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *In Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, pages 117–145. Springer-Verlag, 2000.

[6] 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.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

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

[9] G. Necula and P. Lee. Efficient representation and validation of proofs. In *Proc. 13th Symp. Logic in Computer Science*, pages 93–104, 1998.

[10] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 142–154, New York, NY, USA, 2001. ACM.

[11] William Pugh. Compressing java class files. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 247–258, 1999.

[12] Z. Shao. Implementing typed intermediate language. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323, Baltimore, Maryland, September 1998.