Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 16 pages

Towards Language-Agnostic Mobile Code

Christian H. Stork [1] Peter S. Housel [2] Vivek Haldar [3]
Niall Dalton [4] Michael Franz [5]

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

The Java Virtual Machine is primarily designed for transporting Java programs. As a consequence, when JVM bytecodes are used to transport programs in other languages, the result becomes less acceptable the more the source language diverges from Java. Microsoft's .NET transport format fares better in this respect because it has a more flexible type system and instruction set, but it is not extensible, and (for example) has no provision for supporting explicit programmer-specified parallelism. Both platforms have difficulty making transported programs run efficiently.

This paper discusses first steps towards mobile code representations that are independent (in the sense that the representation can be appropriately parameterized) of the source language (e.g., Java), intermediate representation (e.g., bytecode), and target architecture (e.g., x86). We call this kind of parameterizable framework *language-agnostic*.

We present two techniques which provide parts of the envisioned language-agnostic functionality. Compressed abstract syntax trees as a wire format provide for a very dense encoding of programs at a high level of abstraction. We show how to parameterize the compression algorithm in a modular fashion with knowledge beyond the purely syntactical level. This leads to the notion of *well-formedness by construction*. The second technique defines the semantics of programs by mapping from abstract syntax trees to a typed *core calculus* representation. Based on this representation it becomes possible to use portable definitions of security policies and to execute programs written in different source languages, even if a more efficient trusted native compiler is not available on the target platform.

[1] Email: cstork@ics.uci.edu
[2] Email: housel@ics.uci.edu
[3] Email: vhaldar@ics.uci.edu
[4] Email: ndalton@ics.uci.edu
[5] Email: franz@ics.uci.edu

1 Introduction

Over the past several years, many different mechanisms have been proposed for transporting *mobile code*. Mobile code allows new capabilities to be added to internet browsers, internet appliances or other embedded applications with a minimum of user intervention, and provides support for transporting autonomous network agents between hosts.

Several mutually competing goals are involved in the design of mechanisms for transporting for mobile code. Among these are:

- *Security.* If mobile code is to be installed without user intervention, there must be mechanisms in place to prevent attacks or security compromises from malicious or poorly-written programs. *Type safety* is an important part of many systems' security guarantees.
- *Target Independence.* Mobile code transported using a target-independent format becomes usable on a wider variety of host processors and operating-system environments, and thereby becomes immune to changes in underlying technology.
- *Execution Efficiency.* Mobile code should run as efficiently (in terms of speed and memory consumption) as possible on the target platform. It may be important on modern architectures to optimize the code for specific target characteristics [16,3].
- *Execution Latency.* Mobile code should also be able to start executing on the target platform with minimal delay, implying efficient translation of the transportable format into an executable on the target machine.
- *Compactness.* Programs should be transmitted in a format that is as compact as possible, especially on low-bandwidth links such as wireless networks or dial-up connections.
- *Language Independence.* A widely-used mobile code transport mechanism should be applicable to a wide variety of source languages.

We hope that the above criteria will lead to wide acceptance of our ideas. Of course, some of these goals might not always be reconcilable, for example, security and execution latency. Should execution of a program start while its remainder is still being loaded and, potentially, violates the required security policy? Note also that we do not list code obfuscation as a goal. Even with a dedicated effort we believe that it is ultimately impossible for a code format to prevent reverse-engineering.

We have embarked on a long-term research project concerned with exploring the tradeoffs among the listed goals. In particular, we want to investigate the entire "design space" of language-agnostic (i.e., source-language and intermediate-language independent) mobile code representations. While increasing flexibility along several dimensions, language agnosticism makes some of the above goals more difficult to achieve, most importantly efficiency

and security.

There are several approaches that a mobile code system can take towards maintaining security guarantees. The most primitive approach is code signing as used by browser plug-ins and Microsoft's ActiveX components. The inherent problem with code signing is that the trust in the certification authority or the code signer might not be justified. Code signing has no provisions against malicious or involuntary mistakes by the code producer. It only provides a measure of accountability. Note that code signing is orthogonal to all the other approaches mentioned below and can therefore always be combined with our approach.

An approach, available for strongly-typed source languages, allows the code receiver to verify that a program typechecks. Sun's latest incarnation of the Java virtual machine, KVM, achieves this verification efficiently with the help of 'stackmap' attributes [27]. These annotations increase the class file size by only 5 percent, but still, reduction class file size remains a future goals because class files *per se* are rather large.

The idea of verification via type checking can be carried much further as shown by Morrisett *et al.* [20]. Their Typed Assembly Language (TAL) provides typing rules that guarantee memory safety, type safety, control flow safety, and even some higher-level source-language guarantees.

The most flexible approach to secure mobile code is proof-carrying code (PCC) [22]. It is more general than TAL since it does not target a specific type-safety policy but utilizes arbitrary proofs. PCC's downside is the overhead incurred by the processing of proofs. Recently, attention has shifted towards finding efficient wire formats for these forms of mobile code and their annotations. Necula and Rahul [21] have shown that proofs can be compressed efficiently using an oracle-based proof compression.

We propose a set of complementary technologies. Our wire format is compressed abstract syntax trees (CASTs), allowing full type-checking and optimizing compilation to take place at the receiver, as well as profile-directed dynamic code improvement [15]. Although stack code is suitable for an interpreter based implementation, many systems require higher performance and hence the use of a compiler. Typically, stack based code is transformed into register based code and then conventional compiler techniques are used to generate code. By having a dense format, which leads to quicker load times, and avoiding this "backwards" compilation step, we can hope to decrease the startup time of the application while providing high quality compilation.[6] Although there is a large part of the compilation task carried out on the client, previous projects [11,15] have found that by using a fast, straightforward code generator leads to acceptable inital code and does not badly affect startup time. As our previous studies [25,26] have already shown, the use of CASTs

[6] We do not target platforms that lack the computing power for compilation. In such cases bytecode interpretation might still be an alternative, but we can as well imagine some kind of "compilation server".

provides for a very dense encoding. Typically the process results in file sizes far smaller than existing bytecode and compressed bytecode programs. As the technique applies to general ASTs, the use of a special representation is not required. This allows the use of language or compiler specific representations, which gain an efficient wire format "for free". It may be argued that transporting source-level ASTs can also decrease the effectiveness of our encoding. For example, ASTs which only allow for structured control flow may be exponentially larger than a represention which allows arbitrary GOTOs for some programs. However, it is well known that it is often simpler and *more efficient* to deal with structured control flow in optimizing and parallelizing compilers. Assuming structured programs allows, for instance, the efficient creation of Static Single Assignment form and Program Dependence Graphs using greatly simplified algorithms. Therefore, allowing only structured ASTs may in fact lead to overall greater performance even though the transported programs may be larger.

Here we present a rough sketch of our compression technique and go into some more detail with respect to using semantic information for denser encodings and how to achieve this in a modular fashion. Currently, we are working on a refinement that guarantees that only type-valid programs can be transmitted using the mobile code transport mechanism thereby further integrating decompression and verification into a single step. Assuming that a valid instance of the format means that the program is type-safe, greatly simplifies verification that the program is safe to execute and does not require multiple traversals of the program as is the case in verification of Java bytecode. Our main desire is to explore the notion of "well-formedness by construction", i.e., to explore the feasibility of devising an encoding that allows only the encoding of well-formed programs, with an associated increase in compactness. Our current version of this scheme is described in section 2.

Because the transmission format is closely related to the source language, this scheme requires a language-specific compiler at the code receiver. This requirement can be eliminated by accompanying the transmitted program with a mapping describing the semantics and type rules of the source language in terms of a language-independent intermediate representation. These universal semantics can also be used to define security policies in a portable fashion. It may also be possible to use or augment this mapping to provide some degree of interoperability between various source languages, however, we have not yet investigated this. This approach is described in section 3.

2 Compressed Abstract Syntax Trees: Towards Safety by Construction

Nowadays almost all code is written in high-level languages—the lowest level language of which is C. When a high level program is compiled into a binary or even bytecode, much of the regular high level structure is lost or, at least,

hard to identify. One could argue that this is the price to be paid for high performance as the compiler introduces irregularities in order to optimize the code. This tradeoff seems acceptable if all target platforms are known in advance. Considering bytecode as our compilation target — as is the case for all of today's mobile code platforms — compilation to this intermediate code format always reduces performance in the sense that the code receiver is deprived of using optimizations specific to the original high-level language. One could even argue that bytecode optimizations obfuscate the high-level structure of the original source program making dynamic recompilation harder (but without protecting from reverse engineering).

Here we propose to transport programs as their highly compressed abstract syntax trees. Later on we carry this concept even further by introducing the notion of well-formedness for mobile code. Well-formedness has the nice property that the code receiver can combine decompression and verification into a single pass over the incoming bitstream. Furthermore does this combination lead to even better compression, since context-sensitve information is used to restrict the alternatives from which to chose while decoding.

Our two major contributions are (1) we encode ASTs in a very space efficient manner, and (2) our framework tries to be as generic as possible. In this paper our focus will be on the second point, but we will also summarize how to compress ASTs.

2.1 Why Abstract Syntax Trees?

An AST is a tree representing a source program while abstracting from irrelevant concrete details, e.g., which symbols are used to open or close a block of statements. Every AST conforms to an *abstract grammar* (AG) just as every source program conforms to a concrete grammar. AGs give a succinct description of syntactically correct programs by eliminating superfluous details of the source program. Note also that properties like operator precedence and different forms of nesting are already explicit in the AST's tree structure. The reason why we use ASTs as our preferred wire format (in contrast to general graphs, bytecode, machine language, etc.) for mobile code are:

- ASTs are designed to represent the semantics of a program while maintaining a direct correspondence to the source program. According to Meyer [19] ASTs embody the natural representation domain for programs.
- A certain kind of safety comes for free when using ASTs built according to their abstract grammar. A high-level encoding of programs protects the code consumer against attacks based on low-level instructions, which are hard to control and verify. Even if tampered with, a file in our format guarantees adherence to the AG (and certain semantic constraints) or is invalidated, providing some degree of safety by construction. This is in contrast to bytecode programs, which have to go through an expensive

verification process prior to execution. As an example, the Java language enforces restricted control flow, whereas Java bytecode allows arbitrary gotos, which necessitates verification upon class loading.

- ASTs are better suited to just-in-time (JIT) compilation than bytecode. Compiling down to bytecode results in the loss of most of the high level information contained in source code. One of the first steps of any JIT compiler is to try and recover some of this lost high level information. This can be avoided by shipping ASTs, thus leading to lower compilation latency. This also allows the target compiler to make use of language level constructs to guide advanced optimizations.
- As we show, ASTs can be highly compressed. This is an issue when transferring mobile code over low bandwidth networks (such as wireless ones) or for embedded devices with limited storage. This also leads to an overall gain in performance because the CPU is much faster than disk and network access, and the time saved during disk or network access can be used for decompression and compilation.
- We will show that an AST based format for mobile code can be effectively parameterized by the grammar of the source language, enabling our framework to be used with multiple languages. The next section shows that we carry this parameterization even further for standard context-sensitive constructs.
- ASTs offer a natural way to annotate nested constructs by attributing inner nodes.

2.2 Well-formed ASTs

Since programs must fulfill many context-sensitive requirements it is impossible to device a (context-free) AG such that every AG-conforming AST represents a valid program. Here "valid program" means a program that obeys all static semantic constraints specified in the source language's specification. Examples for static semantic constraints are lexical scoping of local variables or matching of parameter types with the declared function or method signature. Practically speaking, valid programs compile flawlessly without generating any error (or warning) messages.

We call an AST *well-formed* if it obeys its AG and a given set of static semantic constraints. Currently, we aim at turning more and more semantic properties of programming languages into well-formedness properties of the mobile-code format itself. This requires a more expensive encoding/decoding process but alleviates the need for separate verification passes. Furthermore, every semantic constraint, which has been integrated into the compression model, increases the density of the encoding, since it reduces the number of alternatives, which need to be considered by the arithmetic coder.

Normally, semantic properties need to be verified with an extra verification

pass at the code receiver's end incurring additional overhead on mobile code handling. We propose to construct the AST's encoding in such a manner that, ideally, the encoding of well-formed programs *exclusively* is possible. Of course, this increases the complexity of the encoder/decoder considerably and this is precisely the tradeoff we wish to explore here.

It is important that the mechanism for encoding ASTs be generic and extensible, i.e., it should not be restricted to encoding ASTs of only one particular language. Given the grammar of a language, and ASTs of programs in that language, the encoding mechanism should work with any language. In addition to their AG, commonly used languages have rather complicated static semantic constraints. Some of these constraints are alike among many languages and can be implemented using our framework, other constraints might fall out of the range of a common framework. This is subject to further research. Note that much of the constraint checking can be simplified by transforming programs into highly canonical ASTs. For example, the ASTs we use for Java already include additional information from source code analysis, such as full qualification and overloading resolution. In section 2.4 we present several modularization techniques for managing the incurred demand for complex constraints.

2.3 Compressing Abstract Syntax Trees

Compactness is an issue when code is transferred over networks limited in bandwidth, particularly wireless ones. It is also becoming increasingly important with respect to storage requirements, especially when code needs to be stored on embedded devices. Processor performance has increased exponentially over storage access time in the last decade. It is therefore reasonable to investigate compression as a means of using additional processor cycles to decrease the demand on storage access, leading to a net gain in performance. As shown by Franz [11], the time saved for transmission (or file access) can pay for the additional decompression and compilation effort when transporting code as ASTs.[7]

Here we give a brief overview of our encoding of ASTs. For full details see [26,25]. Since the AST conforms to a given abstract grammar (AG), we are using domain knowledge about the underlying language to achieve a more compact encoding than a general-purpose compressor could achieve. Note that both the producer and consumer have knowledge of the grammar. We compress the AST of a program using a novel statistical approach. We apply a text compression technique called *predition by partial match* (PPM) to trees. PPM models the statistical properties of ASTs by maintaining a set of contexts (i.e., paths form the root to a node). Each context maintains the counts of

[7] By now the consensus seems to be that on-the-fly compilation is preferable over bytecode interpretation. For example, in Microsoft's .NET architecture, code in intermediate language format is never interpreted but always compiled.

symbols that followed that context. Based on these counts predictions can be made next time this context is encountered.

Our compression framework has the following conceptual stages:

- *Parsing*: Produce an AST from the source code (i.e. a canonicalization of the parse tree).
- *Serialization*: The AST needs to be converted into a linear sequence of symbols before being encoded. We traverse the tree in depth-first in order to serialize it.
- *Modeling*: The symbols of the serialized AST are encoded in a predictive manner, i.e. a probability of occurrence is assigned to every symbol which may occur next. The grammar and additional semantic constraints plays an important role in this step, because it is used to limit the number of symbols which may possibly occur next.
- *Arithmetic Coder*: The arithmetic encoder [32] uses the model built by the previous step to output an actual stream of bits.

At the consumer's end the AST is incrementally rebuilt in the order it was traversed at the producer's end. The information being used by the modeler at the producer's end is limited to the symbols seen in the traversal up to the one currently being encoded. Thus the coder and modeler at the consumer's end operate conceptually in lock-step with those at the producer's end.

2.3.1 Compressing Constants

A sizable part of an average program consists of constants like integers, floating-point numbers, and, most of all, string constants and identifier names. An efficient way of encoding constants is to specify an index into a table which contains all constants. The number of bits of this index can be further reduced by maintaining different tables for different kinds of constants, such as strings, type names, identifiers and so on. Our prototype stores constants as references into compressed tables of different kind of constants, which we call *global pools*.

2.4 Language Independent Framework

Our framework for compression of ASTs is parameterized by the grammar of the language being parsed. To use our framework with a language, one need only specify its abstract grammar—as a bare minimum. Further context-sensitive constraints can be added in a modular fashion This genericity is important since we are trading a more complicated decompression routine for a simpler one (e.g., gunzip) and the automation provided by our framework minimizes the amount of potential errors. Currently we have tried our framework with two languages — Java and Oberon. In each case, the framework itself did not need to be modified — we only needed to specify the augmented grammar of the language.

We illustrate the genericity and some of the context-sensitive features of

our framework by compressing a trivial subset of Java as an example. This subset supports integer variables, which need to be declared in scope before constant integers can be assigned to them.

The following shows its concrete EBNF grammar on the left and our corresponding extended AG notation on the right. [8]

```
                                    global integer pool  numbers
                                      compressedby deltacompressor.
                                    scoped string pool  idents
                                      compressedby namelistcompressor.

<Number> is integer constant        Number = INTEGER(numbers).
<Ident> is local variable name      Ident = STRING(idents).
Decl ::= "int" <Ident> ";".         Decl = !Ident.
Assign ::= <Ident> "=" <Number>";". Assign = ^Ident Number.
Stmt ::= Block | Assign | Decl.     Stmt = Block | Assign | Decl.
Block ::= "{" Stmt* "}" .           Block = Stmt* ; scopes idents.
```

First the symbol table entries for numbers and identifiers are specified. Since this cannot be done in EBNF the left hand side is kept informal. The corresponding definitions on the right hand side first declare *pools*, which are then used to store instances of `Number` or `Ident` tokens. Pools provide additional semantic information both for their encoding (e.g., delta compressed) and semantic usage constraints (e.g., scoped). The actual productions follow.

The abstract grammar lacks the terminal symbols present in the concrete grammar. Note that `!Ident` denotes an insertion into the `idents` pool, whereas `^Ident` denotes a lookup in the same pool. In this case, it is not even necessary to encode the names of identifiers but only their identities, e.g., by encoding them as n-th previous in-scope declaration (cf. de Bruijn notation of bound variables in λ-calculus). For better readability, keys for the `idents` pool are expressed as strings, e.g., before encoding as their real name and after decoding as `Ident#`*N*. Global pools as used by `Number` work differently. While encoding, a prescan of the AST gathers all occurrences of integer constants into one table, which is then indexed upon each usage of a `Number`. The constant table is compressed and placed in front of the AST encoding proper. Thus all constants are available to the decoder when any indices are encountered.

2.4.1 Weaver/Yarn Design Pattern

It is natural to implement most of our AST processing with the *visitor* design pattern [13]. Visitors are used to walk the AST and perform different tasks on the tree, for example, gathering all occurring constants or computing the probabilities for the arithmetic coder. Visitors are a good means to separate and recombine different passes over the AST.

[8] The actual productions are written in Python — this is a slightly abstracted notation, but has only minor syntactic differences from the real Python code.

Obviously, the encoder and decoder have to work in a symmetric fashion. We need to ensure that the state that contributes to the encoding can easily be reproduced by the decoder given the information available at every point in time. In our case, the relevant state is everything that contributes to the computation of the probabilities handed down to the arithmetic coder. Conceptually many passes, i.e., visitors, are needed both while encoding and while decoding. For example, one pass initializes the probabilities for each choice, the next pass performs the PPM computations, and the last pass encodes the AST using results provided by the former passes. It is very appealing to use the same visitor for encoding and decoding because this guarantees that symmetric operations are performed simply by code-reuse. However, implementing the different passes of the encoder/decoder as visitors causes one problem: the visitors on the decoding side cannot traverse the whole tree one after the other since it is exactly this tree that is being built. The solution is to interleave the execution of all visitors at each node. We call this mechanism *weaving*. In order to weave visitors it is necessary to change the visitor design pattern. A regular visitor has one visit method per potential host node kind. These methods include the method calls to descend to the host node's children.

When several visitors are supposed to be interleaved, their computations have to be split up around the descent into the host's children. We dubbed these new visitor mutants *yarns* since they are being woven by a weaving visitor, the *weaver*, who provides the actual visitor skeleton. The weaver takes a list of yarns as its argument and visits each node just like a visitor would—executing its yarns in the given order before and after traversing the host's children. The yarns ensure that children are constructed before their traversal starts. Yarns communicate among each other via node attributes—very much the way attribute grammars work.

Yarns have proven to be a very versatile tool to modularize our operations on the AST. The extra effort to split up the calculations into before/after method pairs is well worth it because it increases code reuse and aids correctness by construction.

2.5 Results

We show that ASTs are highly compressible using our framework, which fares better than language-specific compression schemes in spite of being generic. We have tried our framework with two languages — Java and Oberon, but most of our compression experiments are with Java owing to the body of work to compare our results with. For completeness, compression figures reported in [25,26] are reproduced in Tables 1 and 2. Table 1 shows results for individual Java classes, and Table 2 gives results for entire Java packages being compressed as a single unit.

To summarize: our technique compresses 5–50% better than the currently

Class Name	Class File	Gzip	Bzip2	Pugh	CAST	$\frac{\text{CAST}}{\text{Pugh}}$
`ErrorMessage`	305	256(155)	270(154)	209	105	50%
`CompilerMember`	1192	637(327)	641(342)	396	230	58%
`BatchParser`	4939	2037(1301)	2130(1412)	1226	1069	87%
`Main`	11363	5482(3880)	5607(3964)	3452	3295	95%
`SourceMember`	13809	5805(4010)	5705(3950)	3601	2988	83%
`SourceClass`	32157	13663(10398)	13157(7608)	8863	7849	89%

Table 1
File size comparison of compressed AST files (CAST) with class files from `sun.tools.javac` compressed using alternative techniques. The numbers in parenthesis in the Gzip and Bzip2 columns indicate file sizes when the corresponding Java source files (after stripping comments, whitespace etc.) were compressed using the Unix gzip and bzip2 compressors.

Package Name	Jar	Gzip Jar	Bzip2 Jar	Pugh	CAST	$\frac{\text{CAST}}{\text{Pugh}}$
`sun.tools.javac`	36787	32615	30403	18021	14070	78%
`jess`	232041	133146	97852	48331	31083	64%

Table 2
File size comparison of compressed collections of classes from two Java packages.

best Java-specific compression scheme by Pugh[23], who compressed Java bytecode, and 3–8 times smaller than regular Java archives, without any substantial tuning of our framework. Note that compressed ASTs are consistently smaller than compressing the concrete syntax (Java source files) using commonly available compressors such as `gzip` and `bzip2`. Note that we have stripped comments and extra whitespace.

3 Minimizing Language Dependence in a High-Level Representation

The representation level of Java Bytecodes is somewhat mixed, containing many features that are specific to the Java language, as well as some features more typical of a low-level encoding. Overall it has proven to be adequate for transporting code in a number of source languages to a variety of popular machine architectures (as well as direct interpretation), while allowing validation by the code consumer.

However, mismatches in the representation level and the semantics of the representation can result in bad performance for languages other than Java.

Potential sources for problems include insufficient flexibility for object storage (resulting in inefficient use of space); differences in the semantics of subtyping, object dispatch or primitive operations; unavailability of tail-call semantics; and lack of support for parallelism or other specialized language constructs.

Shortly after the JVM was introduced, Shivers and Fahlman [24] proposed solving this problem by providing a mechanism for extending the Java Virtual Machine. The mechanism would allow new language-specific instructions to be added to the JVM for use by compilers. The new instructions would be described using a low-level format such as a Register-Transfer Language (RTL). Due to the unsafe, low-level nature of the extension language, extension programs would become part of the trusted computing base on the code consumer's side, and would therefore require some sort of cryptographic authentication mechanism. This proposal was never implemented, and so there is still no adequate transport mechanism for mobile code in many advanced programming languages.

An alternative method for solving this problem, the *core calculus* approach, is currently being pursued. In the base approach, programs are encoded using a high-level intermediate representation based on the source-language specific abstract syntax tree (AST) as described in the previous section. In addition to transporting the encoded AST, a *mapping* is transported, giving the semantics of the source language AST in terms of a core calculus.

Programs are transmitted using this approach as follows: the code producer uses a traditional compiler front-end to produce an AST for the source program, and then directly encodes this tree for transmission. A mapping describing the semantics of the source language is also transmitted, or obtained from a library of mappings for common languages. The code consumer decodes the original AST and the mapping, then applies the mapping to obtain the original program represented in the core calculus. The consumer then compiles the program from this representation to machine code on the target architecture.

To ensure the safety of this scheme, it is necessary for the core calculus to be a type-preserving one. The representation we have chosen in this line of research is to use the typed lambda calculus, System F_ω. This representation has been studied extensively by the programming language community, and is amenable to compiler analysis using well-known techniques.

Advantages of this scheme include:

- The high-level representation makes it possible to compress the code efficiently.
- Source languages can be added *ad hoc* without user intervention or increasing the size of the trusted computing base.
- Because it is easier to write a mapping file than to develop a full compiler back-end, the system can serve as part of a "language design workbench."
- Because programs are transmitted at a high level, and almost all of the

compilation machinery is present at the code consumer, it becomes possible to provide a convenient API for code-writing programs.

- Because the source program is transmitted using a high-level representation, it remains possible to use a (potentially more efficient) language-specific compiler backend for more commonly used languages.

The system as described so far places the burden of work on the code consumer. If the mapping is changed to map a lower-level code to the core calculus, however, it becomes possible to do variable amounts of optimization before the code is transmitted. This allows the flexibility to choose the best place in the continuum of representation levels for the application at hand.

In fact, the complexity of some language features (such as object and other type systems, for example) might make it necessary for the front-end compiler to transform the source program into a form that more resembles a program in a procedural or functional language. This transformation would make the mapping more tractable for a simple mapping language, and reduce the initial compilation latency for the code consumer.

A wide variety of approaches to mapping object-oriented languages into core calculi are seen in the literature [1,17]. The system's mapping language needs to be expressive enough to allow different approaches to these transformation techniques. Ideally, it should also be easy to learn and "accessible" to a wide variety of potential designers of computer languages.

4 Related Work

The initial research relevant to AST compression was conducted in the 1980's and focused on the reduction of storage requirements for Pascal source files. Cameron [5] was the first to combine tree encoding and arithmetic coding. He assigns fixed probabilities to alternatives appearing in the grammar. Tarhio [28,29] uses PPM to drive the arithmetic coder in a fashion similar to ours and Cheney [6] suggests similar ideas for term compression. Franz [11] was the first to use a tree encoding for mobile code.

Even though seemingly placed in the same application domain, research on "code compression" [9,12,18,7] is generally not comparable to the above line of work on source text and AST compression. The reason is that code compression focuses much more on the specifics of machine code like choice of op codes, operand formats, lack of apparent high-level structure, and so on. Nevertheless, some overlap does exist. For example, regularities of ASTs that we exploit adaptively, Evans and Fraser [10] exploit in a static manner.

With respect to Java, Pugh [23] achieves the best compression. His tool compresses Java archives by a factor of 2 to 5. In contrast to Pugh, who essentially compresses class files including their bytecode, Eck *et al.* [8] employ a compression similar to ours. They report compression results similar to Pugh's, although more detailed information is needed to assess their approach.

Necula and Rahul [21] have recently looked at the the problem of devising efficient wire formats for proof carrying code. By viewing a proof as a tree, they achieve good compression by limiting the further possibilities at any given node, and then transmitting only the index of the choice actually taken. This is similar in spirit to the way we compress abstract syntax trees, but uses neither probabilistic predictive compression nor a grammar.

We are not aware of any design pattern like weaver/yarn. Outside the realm of object-oriented programming, the functional language community has developed techniques for the related problem of composing semantic functions on ASTs. They use higher-order functions (parser combinators) to implement grammar constructs such as sequencing and choice [4,31,14]. In this setting the idea of weaving visitors like yarns is equivalent to fusing computations (i.e., *folds*) on the AST, thereby alleviating the need for intermediate data structures (*deforestation*).

The ASF+SDF system [30] allows the syntax and semantics of programming languages to be specified using algebraic specifications. These specifications have been used to generate various language tools but have not as yet been applied to mobile code transport.

Our group is exploring an alternative mobile-code representation, SafeTSA [2], that provides security guarantees like the JVM (and more) expressed statically as well-formedness properties of the encoding itself. Even though SafeTSA stays at the level of abstraction of Java Bytecode it is more suitable for optimization and compression.

5 Conclusions and Future Work

We have argued for compressed abstract syntax trees as a parametrizable wire format. Several ideas for the parametrization and the implementation of security guarantees have been presented. A first prototype implementation shows promising results.

We are currently working on extending our work in several directions, mostly focusing on further improving compression and modularization. To improve compression, we intend to specialize the compression of constants and to further explore the spectrum of models to drive the arithmetic coder. We are also implementing a core calculus mapping system and experimenting with the design of the mapping specification language.

Acknowledgements: We are grateful to the anonymous referees for their comments, which greatly improved this paper. Particular thanks go to Greg Morrisett for reviewing drafts and providing detailed feedback.

References

[1] Abadi, M. and L. Cardelli, "A Theory of Objects," Springer, New York, 1996.

[2] Amme, W., N. Dalton, J. v. Ronne and M. Franz, *SafeTSA: A type safe and referentially secure mobile-code representation*, in: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[3] Benitez, M. E. and J. W. Davidson, *The advantages of machine-dependent global optimization*, in: *PLSA'94: International Conference on Programming Languages and Architectures*, 1994, pp. 105–123.

[4] Burge, W. H., "Recursive Programming Techniques," Addison-Wesley, 1975 .

[5] Cameron, R. D., *Source encoding using syntactic information source models*, IEEE Transactions on Information Theory **34** (1988), pp. 843–850.

[6] Cheney, J., *Statistical models for term compression*, in: *Data Compression Conference*, 2000, p. 550.

[7] Debray, S., W. Evans and R. Muth, *Compiler techniques for code compression*, in: *Workshop on Compiler Support for System Software*, 1999.

[8] Eck, P., X. Changsong and R. Matzner, *A new compression scheme for syntactically structured messages (programs) and its applications to Java and the Internet*, in: *Data Compression Conference*, 1998, p. 542.

[9] Ernst, J., W. Evans, C. W. Fraser, S. Lucco and T. A. Proebsting, *Code compression*, in: *Proceedings of the ACM Sigplan '97 Conference on Programming Language Design and Implementation*, 1997, pp. 358–365, published as Sigplan Notices, 32:5.

[10] Evans, W. and C. Fraser, *Bytecode compression via profiled grammar rewriting*, ACM SIGPLAN Notices **36** (2001), pp. 148–155.

[11] Franz, M., "Code-Generation On-the-Fly: A Key to Portable Software," Ph.D. thesis, ETH Zurich (1994).

[12] Fraser, C. W., *Automatic inference of models for statistical code compression*, in: *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1999.

[13] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, Massachusetts, 1995.

[14] Hutton, G., *Higher-order functions for parsing*, Journal of Functional Programming **2** (1992), pp. 323–343.

[15] Kistler, T., "Continuous Program Optimization," Ph.D. thesis, University of California, Irvine (1999).

[16] Kistler, T. and M. Franz, *Automated data-member layout of heap objects to improve memory-hierarchy performance*, ACM Transactions on Programming Languages and Systems (2000).

[17] League, C., V. Trifonov and Z. Shao, *Type-preserving compilation of featherweight java*, in: *8th Foundations of Object-Oriented Languages Workshop (FOOL'8)*, London, 2001.

[18] Lucco, S., *Split stream dictionary program compression*, in: *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.

[19] Meyer, B., "Introduction to the Theory of Programming Languages," PHI Series in Computer Science, Prentice Hall, 1990.

[20] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to Typed Assembly Language*, ACM Trans. Prog. Lang. and Sys. **23** (1999), pp. 528–569.

[21] Necula and Rahul, *Oracle-based checking of untrusted software*, in: *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.

[22] Necula, G. C., *Proof-Carrying Code*, in: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, 1997.

[23] Pugh, W., *Compressing java classfiles*, in: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999, pp. 247–258.

[24] Shivers, O., *Supporting dynamic languages on the Java virtual machine*, in: *Proceedings of the Dynamic Objects Workshop*, Boston, 1996.

[25] Stork, C. H. and V. Haldar, *Compressed abstract syntax trees for mobile code*, in: *Workshop on Intermediate Representation Engineering (IRE 2001)*, Orlando, Florida, 2001, to appear in Intl. Conf. on Systemics, Cybernetics and Informatics (SCI).

[26] Stork, C. H., V. Haldar and M. Franz, *Generic adaptive syntax-directed compression for mobile code*, Technical Report 00-42, Department of Information and Computer Science, University of California, Irvine (2000), revised April 2001.

[27] Sun Microsystems, "Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices: White Paper," (2000).
URL `http://java.sun.com/products/cldc/wp/`

[28] Tarhio, J., *Context coding of parse trees*, in: *Proceedings of the Data Compression Conference*, 1995, p. 442.

[29] Tarhio, J., *On compression of parse trees*, in: *Proc. of Eigth Symposium on String Processing and Information Retrieval (SPIRE 2001)*, IEEE, 2001.

[30] van Deursen, A., J. Heering and P. Klint, "Language Prototyping: An Algebraic Specification Approach," World Scientific, Singapore, 1996.

[31] Wadler, P. L., *How to replace failure by a list of successes*, in: J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201** (1985), pp. 113–128.

[32] Witten, I. H., R. M. Neal and J. G. Cleary, *Arithmetic coding for data compression*, Communications of the ACM **30** (1987), pp. 520–540.