$(MC)^2$: Mobile Code with a Mobile Compiler

Peter S. Housel
Bren School of Information and Computer Science
University of California
Irvine, CA 92697-3425
housel@acm.org

April 18, 2005

Abstract

Existing methods for transporting program code in a type-safe manner either make use of a type-safe virtual machine representation, or transmit machine code along with a verifiable proof of its safety. High-level representations sacrifice potential flexibility in the choice of source languages, whereas low-level representations sacrifice potential portability in the execution environment. We motivate and describe another alternative, that of transmitting a recipe for generating a compiler along with the transported program code.

1 Introduction

In recent years, various ways have been developed for encoding computer programs so that their safety is insured. The most well-known of these is the Java Virtual Machine [Gos95], which defines a virtual processor that is only capable of "safe" operations. Java source programs are compiled into the language of the Java Virtual Machine for transport to the user's system. Once received, a JVM program is verified to insure that it is well-formed, and then executed, either by interpreting the virtual machine code or by re-compiling it to native machine code using a Just-In-Time (JIT) compiler. The Microsoft Intermediate Language (MSIL) virtual machine operates in a similar manner for C# and other similar languages.

An additional benefit of representing code using a virtual machine is that the code potentially becomes portable to a number of different platforms. Code made portable in this manner is sometimes called *mobile code*.

There are other methods for insuring the safety of transported programs. *Proof-Carrying Code* [Nec98] transports programs in native machine code, along with a logical proof of their safety (at least partially constructed by the compiler). The code recipient then checks the proof (in considerably less time than was required to construct it) before executing the program. *Typed Assembly Language* [MCGW98, MWCG99] takes a similar approach, but uses a framework based on type systems rather than on logical formalisms. Shao *et al* [STSP05] describe another type-theoretic framework for maintaining safety proofs through the compilation and code transport process.

Slim Binaries is another approach to solving both the safety problem and the platform portability problem. Instead of transporting compiled code, a slim binary program consists of a compressed representation of the original source program. There are several potential advantages to this approach:

Compression Compressed abstract syntax trees [FHKS02] can be stored and transmitted using less space than other formats (such as JVM byte code or compressed JVM byte code).

Guaranteed-valid static semantics Dictionary-based compression methods operate by maintaining tables of frequently-occurring strings in the source text, and transmitting dictionary lookup indices instead of the text itself. When transmitting compressed abstract syntax trees, the source programming language's static semantic rules (variable scoping rules, type compatibility rules, and the like) constrain the set of dictionary entries valid at any given point in the source text. Using these constraints during dictionary construction, we can insure that only valid programs (according to the static semantics) can be transmitted. This eliminates the need for a validation step on the receiving end.

Information for just-in-time compilation Just-In-Time compilation allows program optimization to take into account the current dynamic behavior of the program, as well as aspects of the specific target machine (such as architecture variant, cache size, and cache line size) on which the code will run. If a low-level intermediate language is used to transmit the program, some information about the program useful for optimization may require a great deal of effort to reconstruct, and some may be unavailable. When compressed abstract syntax trees are transmitted, however, all necessary information can be computed by the compiler, and preserved for profile-based re-optimization.

A disadvantage of such a high-level representation is that the implementation of the compiler or interpreter used by the code consumer is strongly coupled to the implementation details of the code producer. At the very least, a code representation based on the original source code is not immune to changes in the source language. Additionally, dictionary compression making use of the static semantics makes the code consumer sensitive to changes in any part of the defined semantics accepted by the source compiler; this would make it necessary to redeploy the receiver if the semantics (and therefore the transmitter) is changed in even minor ways.

While virtual machine code representations are less sensitive to such changes, they are not immune. Furthermore, while a single virtual machine representation can be used to express code generated from a number of different source languages, most virtual machine representation are designed with a specific language or class of languages in mind. This limits the suitability of most virtual machine representations for expressing languages that differ in, for example,

- Strict vs. lazy evaluation
- Object model (single vs. multiple inheritance, class-based or prototype-based, object methods vs. multi-methods)
- Other type system and object representation aspects (boxed vs. unboxed primitive values, etc.)
- Exception-handling model
- Space complexity of tail recursion [Cli98]

These restrictions serve to make most virtual machine representations into Procrustean beds that limit the efficiency of programs written in languages that differ significantly from the representation's anticipated use.

2 Mobile Compilers

One approach to making such systems more flexible is to actually transport the just-in-time compiler to the client system. This can take a number of different forms:

Virtual machine extension instructions In this scheme, an ordinary virtual machine representation is used, with a portion of the virtual

instruction space left undefined. The meaning of these instructions is determined by a plug-in extension component for the just-in-time compiler.

Transportable compiler plug-ins Plug-ins can also be used to perform the entire just-in-time compilation task. In this case the plug-in is responsible for the interpretation of the entire stream of transported code.

Compiler generation Instead of transporting a just-in-time compiler, we can instead transport a semantic description from which a compiler can be generated. This idea is the primary topic of this research, and is explored further below.

3 Firewalling Just-In-Time Compilers

Plug-ins can be a source of unsafe or malicious code, and compiler plug-in extensions doubly so. In addition to insuring that a just-in-time compiler plug-in itself does not perform any unsafe operations, it is also necessary restrict the compiler from generating unsafe or malicious target code when it compiles the transported program.

The safety of the plug-in itself can be guaranteed by transporting it using a virtual machine or other safe representation. To ensure the safety of the generated code, however, it is necessary to *firewall* the plug-in by treating its output as potentially unsafe, requiring that it supply a verifiable safety proof before executing it. One way of doing this is to generate a Proof-Carrying Code or Typed Assembly Language proof along with the target object code.

PCC and TAL verifiers are generally smaller and simpler programs than just-in-time compilers. Therefore, even in situations where the just-in-time compiler is not transported, this technique can be used to reduce the size of the *trusted computing base* (TCB), the set of code that needs to be audited to ensure confidence in its safety.

4 Semantic Descriptions

Formal semantics makes use of mathematical formalisms to describe the meaning of computer programs and of computer programming languages. Formal definitions of programming-language semantics are used as a way for language designers to unambiguously communicate programming language

designs to language implementers and users, and for generating language implementations. They can also be used as part of a formal system for proving the correctness of programs written in the language.

A large number of programming language semantic frameworks have been designed, though only a small number of these have had any kind of widespread practical use. Following Mosses [Mos01], and Zhang and Xu [ZX04], these frameworks can be roughly categorized as follows:

Operational Semantics Operational semantic frameworks model program semantics as a set of intermediate states that represent a possible way for arriving at the program's result. Given the abstract syntax tree of a program as the starting state, the rules of an operational semantic description tell how to transition between intermediate states until the program terminates.

The best-known variants of operational semantics are Plotkin's *Structured Operational Semantics* (SOS); Kahn's *Natural Semantics* (otherwise known as "big-step" operational semantics); and Gurevich's *Abstract State Machines*.

Denotational Semantics Denotational semantic frameworks model the *denotation* of programs and parts of programs using abstract mathematical functions, often using typed λ-notation. In addition to the original Scott-Strachey form of denotational semantics, frameworks in this category also include Bjørner and Jones's *VDM Semantics* and Moggi's *Monadic Semantics*.

Axiomatic Semantics Axiomatic semantic frameworks model programs via boolean formulas asserting constraints. The axiomatic *Hoare logic*, *Algebraic Semantics*, and *Dynamic Logic* frameworks are generally considered to be more appropriate for correctness proofs and static checking than for compiler generation.

Hybrid approaches Various approaches exist that combine two or more of the above approaches, such as *Action Semantics*, *Algebraic Operational Semantics*, *Modular Monadic Semantics*, and *Modular Monadic Action Semantics*.

5 Action Semantics

Action Semantics [Mos92] was developed by Peter Mosses and David Watt to be a pragmatically useful language for describing the semantics of "real"

programming languages. While Action Notation is similar in many ways to denotational semantics, mapping programs and program fragments to denotations, it makes use of combinators called *actions* which can be viewed operationally. Action Notation is more accessible than other semantic notations; the intent of the designers was to make the same semantic specifications useful to language designers and language implementors, and understandable by programmers.

Specifications in action notation have been developed for several practical programming languages, including Pascal [MW93], Standard ML [Wat97b], and the Java subset JOOS [Wat97a]. Action Semantics was also used in the definition of ANDF [HT94].

Action notation is designed to be modular, allowing different language specifications to be composed of common "building-block" modules when they share common language constructs [DM03].

The original action notation (AN-1) is defined within an algebraic framework known as Unified Algebra [Mos89]. A revised form of the notation, AN-2, is defined within the CASL [Mos00] algebraic framework.

6 Generating Compilers from Action Notation Specifications

For our purposes, the most interesting use for formal programming language semantics is the automatic generation of language implementations. In particular, our primary goal is to be able to transport language compilers to target machines by shipping semantic descriptions rather than executable code. Such semantic descriptions would be completely target machine-independent, and cannot themselves execute malicious code.[1] The high-level, declarative nature of semantic descriptions would also allow designers of domain-specific and general-purpose programming languages to use the system part of a "language-design workbench," allowing them to prototype the new language semantics without having to develop code generators.

Previous research projects in the late 1980s to mid 1990s have developed semantics-directed compiler generators based on Action Semantics.

[1] Descriptions that amount to "denial-of-service" attacks, where the input program is transformed into endlessly-growing intermediate forms, may still be possible.

6.1 Peter Lee's MESS System

Peter Lee's MESS system [Lee89] generated a compiler front end and "compiler core" from specifications of the context-free syntax and semantics of a programming language. The semantic framework used, called *high-level semantics*, was developed by Pleban and Lee [LP87] in order to overcome various shortcomings of traditional denotational semantics. Inspired by the approach taken by action semantics, high-level semantics divides the semantic specification into macro-semantic and micro-semantic levels, with the micro-semantic specification serving to define the semantics of the macro-semantics. Differing micro-semantic specifications (with the same interface signature) could be used to generate compilers producing various low-level codes (which could then be interpreted), or used to construct a code generator.

6.2 Jens Palsberg's Cantor System

Jens Palsberg's CANTOR system [Pal92] generates a compiler (generating SPARC assembly code) from a programming language specification in action notation. The CANTOR compiler generator, written in Perl, read a language specification (in a LaTeX-syntax form of action notation) and generated a compiler (as a Scheme program) that accepted an abstract syntax tree as a Scheme S-expression, expanded the program into an action denotation, and generated assembly code from the denotation. Run-time speed of the generated code was about two orders of magnitude slower than that of an ordinary compiler.

The correctness of the entire system was proven (by hand) within the Unified Algebras framework.

6.3 Bondorf and Palsberg's System

Bondorf and Palsberg [BP93] developed a compiler for actions using the first Futamura projection, applying a partial evaluator (Similix) to specialize an interpreter for program denotations in action notation. This yielded a compiler which took an expanded action (the result of applying a language specification to an abstract syntax tree) to produce compiled output in the form of a Scheme program. The compile time of compilers generated this system was considerably better than that of CANTOR, though the generation of the compiler took much longer. Performance of the generated code was comparable to that of CANTOR output.

6.4 Peter Ørbæk's Oasis System

Peter Ørbæk's OASIS system [Ørb93, Ørb94] was designed as an improvement over Palsberg's CANTOR. While similar in structure, OASIS incorporated additional analysis phases to annotate the expanded actions and enable the code generator to produce more efficient code. As a result, the execution time of the generated code was within a factor of 1.7 compared to the output of `gcc` (without optimizations).

6.5 Kyung-Goo Doh's Transformation

In [Doh95], Kyung-Goo Doh describes a two-level type analysis which can be used to determine what parts of an action can be statically determined at compile time. This analysis can be used to partially evaluate language specifications, eliminating unnecessary actions from the output of generated compilers.

6.6 Brown, Moura, and Watt's Actress System

The ACTRESS system [BMW92, MW94, Bro92] consists of a number of modules for analyzing actions, performing transformations on them, and generating C code. The transformations implemented by ACTRESS are said to be better at eliminating unnecessary transient values and bindings than Doh's method. Run-time performance of ACTRESS output is not as good as that of OASIS-generated programs, largely because it does not incorporate the same degree of traditional optimizations specifically geared towards generation of efficient code.

6.7 Kent Lee's Genesis system

Kent Lee's Genesis compiler generator [Lee00, Lee99] has a similar structure to the other systems. Before generating code, Genesis compilers transform the action denotation intermediate representation into a subset of full action notation called *postfix actions* which more closely matches the characteristics of a stack machine or RISC architecture. The result of this transformation is used to generate JVM byte codes.

7 Research Program

As outlined above, the overall goals of this research program are to make possible a mobile code transport system that:

- Transports programs in a compressed source format, but does not require redeployment when the source language evolves.
- Does not "ghettoize" languages that are not close matches to the ones in the narrow design space for which it was designed.
- Preserves a guarantee of type safety for compiled programs.

Towards that end, it will concentrate on the following areas.

7.1 Automatic Compressed AST Generation

For the most part, compiler generation from semantic definitions is concerned with implementing the *dynamic semantics* of a language, i.e., modeling the computation of the program. However, semantic frameworks such as action notation are also capable of expressing static (compile-time) semantics. Doh and Schmidt [DS92] discuss a method for extracting static type inference rules from an action semantic specification. Watt's semantic specification of Standard ML [Wat99] includes both static and dynamic semantics.

The inclusion of static semantics in a language specification and the automatic extraction of typing rules from such a specification can be applied to the problem of keeping the producer and consumer of compressed abstract syntax trees synchronized. If the AST encoder, decoder, and compiler are generated from the same semantic definition it can be assured that they will always construct the same compression dictionaries on the transmitting and receiving ends.

7.2 Generation of Just-In-Time Compilers

As discussed above, previous work [Ørb93, MW94, Lee99] has demonstrated the implementation of various optimizations and transformations on programs represented as action denotations. The research program will investigate which of these transformations are cost-effective in the context of a just-in-time compiler, and identify others. It will also identify transformations that can be performed as the JIT compiler is being generated, rather than during the compilation of generated programs.

7.3 Preservation of Type Safety Proof Information

It is anticipated that a mobile code transport system and JIT compiler based on semantics-based compiler generation will require a large trusted

code base, potentially larger than that of common JIT compilers for the Java Virtual Machine [AW02]. The research program will investigate techniques for generating and preserving type safety proof information in the generated just-in-time compilers, allowing the use of a "JIT compiler firewall" to minimize the size of the TCB.

7.4 Representation Level Transformations

Because the meaning of the transported code is determined by the associated semantic description, the representation can be used to express the entire continuum of high-level to low-level representations. This means that an offline tool could, for example, take a program in a high-level language (along with a semantic specification of the language in which it is written) and produce a pre-optimized low-level representation of the program, along with a new semantic description for the low-level language used.

References

[AW02] Andrew W. Appel and Daniel C. Wang. Jvm tcb: Measurements of the trusted computing base of java virtual machines. Technical Report TR-647-02, Princeton University Department of Computer Science, 2002.

[BMW92] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *LNCS*, pages 95–109. Springer-Verlag, 1992.

[BP93] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *FPCA'93, Proc. Sixth ACM Conf. on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 308–317, 1993.

[Bro92] Deryck F. Brown. *Sort Inference in Action Semantics*. PhD thesis, University of Glasgow, 1992.

[Cli98] William D. Clinger. Proper tail recursion and space efficiency. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 174–185. ACM Press, 1998.

[DM03] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47:3–36, 2003.

[Doh95] Kyung-Goo Doh. Action transformation by partial evaluation. In *PEPM'95, Proc. ACM/SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Transformation, La Jolla, California*, pages 230–240, 1995.

[DS92] Kyung-Goo Doh and David A. Schmidt. Extraction of strong typing laws from action semantics definitions. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *LNCS*, pages 151–166. Springer-Verlag, 1992.

[FHKS02] Michael Franz, Vivek Haldar, Chandra Krintz, and Christian H. Stork. Tamper-proof annotations by construction. Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, 2002.

[Gos95] James Gosling. Java intermediate bytecodes: Acm sigplan workshop on intermediate representations (ir'95). In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pages 111–118. ACM Press, 1995.

[HT94] Bo Stig Hansen and Jens Ulrik Toft. The formal specification of ANDF, an application of action semantics. In Peter D. Mosses, editor, *AS 1994: Proceedings of the First International Workshop on Action Semantics*, number NS-94-1 in BRICS Notes Series, pages 34–42, Edinburgh, Scotland, April 1994. BRICS, Dept. of Computer Science, Univ. of Aarhus.

[Lee89] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, 1989.

[Lee99] Kent D. Lee. *Action Semantics-based Compiler Generation*. PhD thesis, University of Iowa, 1999.

[Lee00] Kent D. Lee. Postfix transformations for action notation. In Mosses and de Moura [MdM00], pages 37–46.

[LP87] Peter Lee and Uwe Pleban. A realistic compiler generator based on High-Level Semantics. In *POPL'87, Proc. 14th Ann. ACM Symp. on Principles of Programming Languages*. ACM, 1987.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28–52. Springer-Verlag, 1998.

[MdM00] Peter D. Mosses and Hermano Perrelli de Moura, editors. *AS 2000: Proceedings of the Third International Workshop on Action Semantics*, number NS-00-6 in BRICS Notes Series, Recife, Brazil, May 2000. BRICS, Dept. of Computer Science, Univ. of Aarhus.

[Mos89] Peter D. Mosses. Unified algebras and action semantics. In *STACS'89, Proc. Symp. on Theoretical Aspects of Computer Science, Paderborn*, volume 349 of *LNCS*. Springer-Verlag, 1989.

[Mos92] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Mos00] Peter D. Mosses. CASL and Action Semantics. In Mosses and de Moura [MdM00], pages 62–78. Available also at `http://www.brics.dk/~pdm/papers/Mosses-AS-2000-CASL+AS/`.

[Mos01] Peter D. Mosses. The varieties of programming language semantics. In *PSI 2001: Perspectives of System Informatics 4th International Andrei Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*, pages 165–190, 2001.

[MW93] Peter D. Mosses and David A. Watt. Pascal action semantics, version 0.6. `http://www.brics.dk/~pdm/papers/MossesWatt-Pascal-AS/`, March 1993.

[MW94] Hermano Moura and David A. Watt. Action transformations in the Actress compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *LNCS*, pages 16–30. Springer-Verlag, 1994.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *Transactions*

on Programming Languages and Systems, 21(3):527–568, May 1999.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[Ørb93] Peter Ørbæk. Analysis and optimization of actions. Master's thesis, University of Aarhus, 1993.

[Ørb94] Peter Ørbæk. OASIS: An optimizing action-based compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *LNCS*, pages 1–15. Springer-Verlag, 1994.

[Pal92] Jens Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *ICCL'92, Proc. Fourth IEEE Int. Conf. on Computer Languages, Oakland*, pages 117–126. IEEE, 1992.

[STSP05] Zhong Shao, Valery Trifonov, Bratin Saha, and Nikolaos Papaspyrou. A type system for certified binaries. *Transactions on Programming Languages and Systems*, 27(1):1–45, 2005.

[Wat97a] David A. Watt. JOOS action semantics. FTP[2], October 1997.

[Wat97b] David A. Watt. Standard ML action semantics. FTP[3], May 1997.

[Wat99] David A. Watt. The static and dynamic semantics of SML. Number NS-99-3 in BRICS Notes Series, Amsterdam, The Netherlands, May 1999. BRICS, Dept. of Computer Science, Univ. of Aarhus.

[ZX04] Yingzhou Zhang and Baowen Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Notices*, 39(3):14–30, 2004.

[2] http://www.dcs.gla.ac.uk/~daw/research-projects/JAS.html
[3] http://www.dcs.gla.ac.uk/~daw/research-projects/AAS.html