Bytecode Verification for Haskell

Robert Dockins and Samuel Z. Guyer

Department of Computer Science
Tufts University

Abstract

In this paper we present a method for verifying Yhc bytecode, an intermediate form of Haskell suitable for mobile code applications. We examine the issues involved with verifying Yhc bytecode programs, and we present a proof-of-concept bytecode compiler and verifier.
Verification is a static analysis which ensures that a bytecode program is type-safe. The ability to check type-safety is important for mobile code situations where untrusted code may be executed. Type-safety subsumes the critical memory-safety property and excludes important classes of exploitable bugs, such as buffer overflows. Haskell's rich type system also allows programmers to build effective, static security policies and enforce them using the type checker. Verification allows us to be confident the constraints of our security policy are enforced.

1 INTRODUCTION

The Haskell programming language is uniquely suited to be a host language for mobile code execution. Haskell is both *type-safe* and *purely-functional*. Type-safety is a desirable property because it eliminates entire classes of exploitable bugs, such as buffer overflows. Furthermore, because Haskell is purely functional, side-effects are only possible via a special mechanism called the IO monad. Pure programs (those not "in" the IO monad) are utterly benign. Pure computations can only do three things: return a value; throw an exception; or fail to terminate. They cannot perform potentially malicious actions, such as altering files or communicating over a network.[1]

These two properties complement each other especially well. The IO monad is constructed in such a way that every program fragment which performs side-effects has a type which mentions the IO type constructor. This means that it is possible to statically determine if a program fragment will cause side-effects simply by examining its type. A mobile code execution system can thus rely on Haskell's type system to distinguish code which is safe to run from code which might not be.

Although preventing untrusted code from directly performing side effects sounds quite limiting, Haskell's general framework for monadic code allows us to create a *safe* IO-like monad which we can use instead. Code in this restricted monad can then be executed in a trusted kernel which makes IO monad calls on behalf of the untrusted client if the requested actions are allowed by a security policy.

[1] Many Haskell implementations allow exceptions to this rule via unsafe primitive "functions." We must additionally restrict access to such unsafe primitives to achieve unconditional safety.

Note that using a custom-designed IO proxy monad implements a default-deny style of security policy; anything not specifically allowed by the proxy monad simply cannot be done. Contrast this approach with that of Java, in which "sensitive" operations in the base library must be instrumented with permissions checks. If an omission or misjudgment is made in the placement of these security checks, then a security compromise can occur. In our approach, however, omissions instead result in an inability to perform desired operations. We claim this is a superior failure mode because it represents an easily-visible error which is likely to be quickly corrected. In contrast, the failure mode of a default-allow policy results in a latent security vulnerability, which may go uncorrected for a significant length of time.

Our proposed custom proxy-monad approach can also be augmented by using "lightweight static capabilities" [**?**]. Using the technique of lightweight static capabilities, programmers can implement flexible, static policies to ensure in-bounds array access, ensure file handles are only used while the file is open, and a variety of other security and correctness invariants. The technique relies on a library author to write a trusted kernel which implements the user-visible API. This API will present only safe operations to the user, even though it may be implemented using unsafe primitives. Parametric polymorphism is used to generate unique names (capabilities) which ensure that API operations are only used in allowed ways.

Sadly, transmitting raw source code is unacceptable for mobile applications. Compilation of source code is an expensive operation which is best performed only once. Additionally, the authors of mobile application code may be unwilling to disclose their raw source code. Instead of using source code, mobile execution systems are usually based on *bytecode*, an intermediate program representation which is designed to be easy to interpret while retaining architecture portability.

In late 2005, the Yhc Haskell project was started. It consists of a bytecode compiler and an independent runtime system which interprets bytecode programs. It currently implements the vast majority of the Haskell 98 language standard. Platform and architecture portability is one of the major goals of the Yhc project. The runtime is written in portable C, has few library dependencies, and a fairly small code text footprint. These factors make the Yhc runtime attractive to reuse as part of a web browser plug-in or applet container application.

Unfortunately, once Yhc compiles a program to bytecode, we can no longer typecheck the program; all type information has been erased during compilation. This means that the runtime has no way to distinguish programs which are safe to run from those which are potentially unsafe. It is easy to hand-construct bytecode programs that violate assumptions made by the runtime system and cause it to crash. It would also be very easy to write programs that perform malicious side-effects, such as installing trojans or setting up spam relays. If we wish to use the Yhc runtime as a platform for mobile code execution, we must find a way to ensure that untrusted code is safe before it is executed so that the executing host is not compromised.

Our proposed solution is to extend the type-safety guarantees provided by the source language onto bytecode programs. In order to do this, we must design a

system which rejects programs that are not type-safe and accepts all well-typed Haskell programs. The process of type-checking bytecode programs is called "verification."

Our major contributions are a type system (based on system F) suitable for type-checking Yhc bytecode programs, a method for encoding type information into a type certificate, a prototype compiler which produces type-certified Yhc programs, and an implementation of a certificate verifier. Our verification algorithm is lightweight and should be suitable for adaptation to limited-resource machines.

Neither type-checking variants of system F nor type-checking stack-based bytecode instruction sets are novel; however, their combination is novel. The major technical innovation required to handle advanced type system features (such as parametric polymorphism and recursive types) is the "type rewrite rule." The technique of type rewrite rules is quite general; new rules can be added to expand the type system provided only that they satisfy a simple property called "validity." We can therefore easily extend our type system by adding new type rewrite rules.

2 TYPE-CERTIFIED COMPILATION

Our goal is to create a system for compiling Haskell source to Yhc bytecode and checking the resulting bytecode for type-correctness. This functionality is best split into two parts: a certifying compiler and a certificate checker. Writing a full Haskell compiler is a large undertaking, so we have simplified our problem by first focusing on an intermediate language which is easier to parse and typecheck than raw Haskell source. This intermediate language is based on the higher-order polymorphic lambda calculus, also known as System F_ω [4, 15]. The intermediate language is capable of representing a significant subset of the Haskell language. We believe that those elements which were excluded (primarily the module system and the IO monad) can be added with moderate effort.

The initial version of our compiler takes this intermediate language as its "source" code and produces executable Yhc bytecode. The Yhc virtual machine is based on the G-Machine, a well-known technique for implementing lazy functional languages [1, 14]. Yhc programs consist of a series of super-combinator definitions, where each super-combinator is defined using a sequence of bytecode instructions.[2]

In this paper, we present a simplified version of full intermediate language (hereafter, the IL). We have omitted the primitive types, type-level products, non-recursive let, and the `error` and `seq` primitives. These features are important for the translation of Haskell source but do not add significantly to the discussion of type verification; we elide them in the interests of brevity. The syntax and semantics of the simplified IL are given in tables 1, 2, and 3 found throughout this section. The full version of the IL is treated in an accompanying technical report [3].

[2] For reasons of space, we cannot include a description of the full bytecode instruction set here. Documentation of the Yhc bytecode set is available from the Yhc website at: `http://www.haskell.org/haskellwiki/Yhc`.

k ::=	Kinds:
$\star$	Base Kind
$k \Rightarrow k$	Arrow Kind
Π_i	Product Kind

t ::=	Types:
A	Type Variable
$\hat{\lambda}\mathrm{A} :: k.\ t$	Type-Level Abstraction
$t\ t$	Type-Level Application
$t \rightarrow t$	Arrow Type
$\forall \mathrm{A} :: k.\ t$	Polymorphic Type
$\mu \mathrm{A} :: k.\ t$	Recursive Type
$\langle\vert\, t_0, \cdots, t_{n-1} \,\vert\rangle$	Product Type
$\{\vert\, i_0 : t_0, \cdots, i_{n-1} : t_{n-1} \,\vert\}$	Sum Type

m ::=	Terms:
x	Term Variable
$\lambda x :: t.\ m$	Abstraction
$m\ m$	Application
$\Lambda \mathrm{X} :: k.\ m$	Type Abstraction
$m\ [t]$	Type Application
$\langle m_0, \cdots, m_{n-1} \rangle$	Product
$\pi_i\ m$	Projection
$\&C_i\,[j_0 : t_0, \cdots, j_{n-1} : t_{n-1}]\ m$	Sum Constructor
$\texttt{roll}[t]\,m$	Recursive Roll
$\texttt{unroll}\,m$	Recursive Unroll
$\texttt{let}\,\{x_0 :: t_0 = m_0;\ \cdots\ x_{n-1} :: t_{n-1} = m_{n-1};\}\,\texttt{in}\,m$	Recursive Let
$\texttt{case}\,m\ \texttt{default}\,m\,\texttt{of}\ \{\ i_0 : m_0;\ \cdots\ i_{n-1} : m_{n-1};\ \}$	Case

Terms which differ only in their bound variables are considered identical. All n-ary syntactic constructs (sums, products, let and case), are allowed at any $n \geq 0$.

$$(\lambda x :: t.\ m)\ l \triangleright [x \mapsto l]m \qquad \text{(E-BETA)}$$

$$\pi_i\ \langle m_0, \cdots, m_{n-1} \rangle \triangleright m_i \qquad \text{(E-PROJPROD)}$$

$$(\Lambda \mathrm{X} :: k.\ m)\ [t] \triangleright [\mathrm{X} \mapsto t]m \qquad \text{(E-TYBETA)}$$

$$\texttt{unroll}\,(\texttt{roll}[t]\,m) \triangleright m \qquad \text{(E-UNROLLROLL)}$$

$$\frac{j_x = i}{\texttt{case}\,(\&C_i\,[\cdots]\ m)\ \texttt{default}\ d\ \texttt{of}\ \{\ j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\ \} \triangleright (a_x\ m)} \qquad \text{(E-CASEARM)}$$

$$\frac{\nexists x\ j_x = i}{\texttt{case}\,(\&C_i\,[\cdots]\ m)\ \texttt{default}\ d\ \texttt{of}\ \{\ j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\ \} \triangleright d} \qquad \text{(E-CASEDEFAULT)}$$

TABLE 1. Simplified syntax of the IL with selected evaluation rules

Our compiler uses a standard compilation pipeline. First, the compiler front-end parses and typechecks the IL "source." The IL AST is manipulated in the middle-end before lowering translates the AST into a linear sequence of instructions. Finally, the back-end performs some additional analyses before emitting the executable bytecode file.

The front-end was designed to be easy to implement. The IL concrete syntax is LL(1) and is as close as possible to a direct visual representation of the abstract syntax. We use the recursive-decent techniques of the Parsec library to implement the parser [6]. The IL is also designed to be easy to typecheck. It therefore requires fully elaborated Church-style type annotations on all variable binders. Using a Church-style typing discipline allows the typechecker to work in a fully top-down manner, which significantly simplifies development.

We are uninterested in program optimization at this point, so the only transfor-

$$\frac{(\mathrm{x},k) \in \Gamma}{\Gamma \vdash_k \mathrm{x} :: k} \quad \text{(K-VAR)}$$

$$\frac{(\mathrm{x},k_1),\Gamma \vdash_k t :: k_2 \qquad \text{KGOOD}(\, k_2\,)}{\Gamma \vdash_k (\forall \mathrm{x} :: k_1.\, t) :: k_2} \quad \text{(K-ALL)}$$

$$\text{KGOOD}(\, \star\,) \quad \text{(KGOOD-STAR)}$$

$$\text{KGOOD}(\, \Pi_i\,) \quad \text{(KGOOD-PROD)}$$

$$\frac{(\mathrm{x},k_1),\Gamma \vdash_k t :: k_2}{\Gamma \vdash_k (\hat{\lambda}\mathrm{x} :: k_1.\, t) :: k_1 \Rightarrow k_2} \quad \text{(K-LAM)}$$

$$(\mu \mathrm{x} :: k.\, t) \looparrowright [\mathrm{x} \mapsto \mu \mathrm{x} :: k.\, t]t \quad \text{(UR-MU)}$$

$$\frac{\Gamma \vdash_k t_1 :: k_1 \Rightarrow k_2 \qquad \Gamma \vdash_k t_2 :: k_1}{\Gamma \vdash_k t_1\ t_2 :: k_2} \quad \text{(K-APP)}$$

$$\frac{t \looparrowright t'}{t\ s \looparrowright t'\ s} \quad \text{(UR-APP)}$$

$$\frac{\begin{array}{l}\Gamma \vdash_k t_1 :: k_1 \qquad \text{KGOOD}(\, k_1\,) \\ \Gamma \vdash_k t_2 :: k_2 \qquad \text{KGOOD}(\, k_2\,)\end{array}}{\Gamma \vdash_k t_1 \rightarrow t_2 :: \star} \quad \text{(K-ARR)}$$

$$(\hat{\lambda}\mathrm{x} :: k.\, t)\ s \triangleright [\mathrm{x} \mapsto s]t \quad \text{(TYE-BETA)}$$

$$\frac{(\mathrm{x},k_1),\Gamma \vdash_k t :: k_2}{\Gamma \vdash_k (\mu \mathrm{x} :: k_1.\, t) :: k_2} \quad \text{(K-MU)}$$

$$\frac{t \triangleright t'}{t\ s \triangleright t'\ s} \quad \text{(TYE-APP)}$$

$$\frac{\Gamma \vdash_k t_i :: \star \qquad \text{for all } 0 \le i < n}{\Gamma \vdash_k \langle\!|\, t_0, \cdots, t_{n-1}\,|\!\rangle :: \Pi_n} \quad \text{(K-PROD)}$$

$$\frac{\begin{array}{c}t_i \triangleright^* \langle\!|\, v_0, \cdots, v_{r-1}\,|\!\rangle \\ \Gamma \vdash_k t_i :: k_i \qquad \text{for all } 0 \le i < n\end{array}}{\Gamma \vdash_k \{\!|\, j_0 : t_0, \cdots, j_{n-1} : t_{n-1}\,|\!\} :: \star} \quad \text{(K-SUM)}$$

In the kinding and typing relations, Γ refers to an environment which binds type variables to their kinds and term variables to their types. In all kind and type judgments, the environment is assumed to be well formed, and we omit uninteresting variable freshness side conditions. The notation $\triangleright^*$ refers to the reflexive, transitive closure of the evaluation relation.

TABLE 2. Kinding, Type Evaluation, and Unrolling Rules for the IL

mation performed on the AST before lowering is the lambda-lifting transformation. Lambda-lifting transforms ordinary function definitions into super-combinators. We used the multi-phase lambda-lifting technique due to Peyton-Jones and Lester, although their techniques had to be adapted to our typed calculus [12].

After lambda-lifting, each super-combinator is lowered to a basic *typed* G-Machine instruction set, called the stage 1 G-Machine. In the stage 1 G-Machine, each data item on the stack conceptually has two parts: the data itself and the type currently assigned to that data. During lowering, each non-computational IL construct is translated into a stage 1 instruction that explicitly applies a type rewrite rule (see the next section for discussion of these rewrite rules). The stage 1 bytecode instructions are given in table 4.

After lowering is complete, we transform stage 1 bytecode into the stage 2 bytecode set. Stage 2 bytecodes are more similar to Yhc's and do not explicitly manipulate types. During this transformation we collect together the type manipulation instructions and segregate them from the computational instructions. These

$$\frac{(x,t) \in \Gamma}{\Gamma \vdash_{ty} x :: t} \quad \text{(TY-VAR)}$$

$$\frac{(\mathrm{X},k),\Gamma \vdash_{ty} m :: t}{\Gamma \vdash_{ty} (\Lambda \mathrm{X} :: k.\ m) :: (\forall \mathrm{X} :: k.\ t)} \quad \text{(TY-TYLAM)}$$

$$\frac{\Gamma \vdash_{ty} m :: t_2 \qquad t_1 \rhd^* t_2}{\Gamma \vdash_{ty} m :: t_1} \quad \text{(TY-EVAL)}$$

$$\frac{\Gamma \vdash_{ty} m :: (\forall \mathrm{X} :: k.\ t) \qquad \Gamma \vdash_k t' :: k}{\Gamma \vdash_{ty} m\ [t'] :: [\mathrm{X} \mapsto t']t} \quad \text{(TY-TYAPP)}$$

$$\frac{\begin{array}{c}(x,t_1),\Gamma \vdash_{ty} m :: t_2 \\ \Gamma \vdash_k t_1 :: k \qquad \mathrm{KGOOD}(\ k\ )\end{array}}{\Gamma \vdash_{ty} (\lambda x :: t_1.\ m) :: t_1 \to t_2} \quad \text{(TY-LAM)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{ty} m :: t \qquad \text{the } j_h \text{ are ordered} \\ \Gamma \vdash_k \{|\ j_0 : t_0, \cdots, i : t, \cdots, j_{r-1} : t_{r-1}\ |\} :: \star\end{array}}{\begin{array}{c}\Gamma \vdash_{ty} \&C_i\,[j_0 : t_0, \cdots, i : t, \cdots, j_{r-1} : t_{r-1}]\ m :: \\ \{|\ j_0 : t_0, \cdots, i : t, \cdots, j_{r-1} : t_{r-1}\ |\}\end{array}} \quad \text{(TY-CON)}$$

$$\frac{\Gamma \vdash_{ty} m_1 :: t_1 \to t_2 \qquad \Gamma \vdash_{ty} m_2 :: t_1}{\Gamma \vdash_{ty} m_1\ m_2 :: t_2} \quad \text{(TY-APP)}$$

$$\frac{\Gamma \vdash_{ty} m_i :: t_i \qquad \Gamma \vdash_k t_i :: \star \qquad \text{for all } 0 \le i < n}{\Gamma \vdash_{ty} \langle m_0, \cdots, m_{n-1} \rangle :: \langle\!|\ t_0, \cdots, t_{n-1}\ |\!\rangle} \quad \text{(TY-PROD)}$$

$$\frac{\Gamma \vdash_{ty} m :: \langle\!|\ t_0, \cdots, t_{n-1}\ |\!\rangle \qquad 0 \le i < n}{\Gamma \vdash_{ty} \pi_i\ m :: t_i} \quad \text{(TY-PROJ)}$$

$$\frac{\Gamma \vdash_{ty} m :: u \qquad u \looparrowright v}{\Gamma \vdash_{ty} \mathtt{unroll}\, m :: v} \quad \text{(TY-UNROLL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_k t :: k \qquad \mathrm{KGOOD}(\ k\ ) \\ t \rhd^* u \qquad u \looparrowright v \qquad \Gamma \vdash_{ty} m :: v\end{array}}{\Gamma \vdash_{ty} \mathtt{roll}[t]\, m :: t} \quad \text{(TY-ROLL)}$$

$$\frac{\begin{array}{c}\Gamma' = (x_0,t_0), \cdots, (x_{n-1},t_{n-1}), \Gamma \\ \Gamma' \vdash_{ty} m :: t \qquad \Gamma' \vdash_k t :: k \qquad \mathrm{KGOOD}(\ k\ ) \\ \Gamma' \vdash_{ty} m_i :: t_i \qquad \Gamma' \vdash_k t_i :: \star \qquad \text{for all } 0 \le i < n\end{array}}{\Gamma \vdash_{ty} (\mathtt{let}\,\{x_0 :: t_0 = m_0; \cdots; x_{n-1} :: t_{n-1} = m_{n-1};\}\,\mathtt{in}\, m) :: t} \quad \text{(TY-RECLET)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{ty} m :: \{|\ i_0 : s_0, \cdots, i_{r-1} : s_{r-1}\ |\} \qquad \Gamma \vdash_{ty} d :: t \\ \text{the } j_y \text{ are ordered} \\ \Gamma \vdash_{ty} a_y :: s_x \to t \qquad \text{forall } x, y \text{ where } i_x = j_y\end{array}}{\Gamma \vdash_{ty} \mathtt{case}\, m\, \mathtt{default}\, d\, \mathtt{of}\, \{\ j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\ \} :: t} \quad \text{(TY-CASE)}$$

TABLE 3. Typing Rules for the IL

segregated type rewrite rules become one of the major components of the type certificate.

After stage 2 bytecode is generated, we perform some additional post-processing and analysis that is required before emitting actual Yhc bytecodes. This processing includes things like calculating maximum stack sizes and generating string tables.

As a sanity check, the last step of the back-end calls the certificate verifier to ensure that the emitted bytecode correctly verifies. We found that verifying code was a helpful way to uncover bugs in the compiler. For example, we discovered an off-by-one error in the code generator because the resultant code would not validate. We were able to quickly trace back the source of the bug by using the error message from the verifier.

Finally, after the type certificate has been verified, the actual Yhc bytecode program is serialized to disk. Currently, there is no file-level format for serializing the type certificate. It would, however, be an easy task to create one. We hope to

explore the possibility of integrating the type certificate into the bytecode program file format itself.

3 CERTIFICATE CHECKING

The Yhc execution model consists of an overall G-Machine structure, with a stack machine instruction set for implementing super-combinators. To verify a bytecode program, we require the following information from the type certificate:

- the definitions of all type synonyms used in the program,
- the type and arity of each super-combinator in the program, and
- for each bytecode instruction in the program, a (possibly empty) list of type rewrite rules to apply.

The heart of the certificate checking algorithm is based on abstract interpretation. We are primarily interested in calculating the "shape" of the local execution stack at each program point. The most important aspect of the stack shape is the number of items on the stack at a given point and the type of data each stack slot contains. For each super-combinator definition, we start with an empty stack at position 0. We then calculate the effect each instruction has on all its next possible program points and record the shape of the stack at these points. If we can fill in all program points with a valid stack shape, the preconditions for each instruction are met, and all control-flow paths through the bytecode stream end by returning a data item of the correct type, then we can conclude that the super-combinator is well-typed.

A nice feature of our algorithm is that it only requires one pass over the bytecode to perform verification. One could even go a step further by pre-calculating the stack shapes and placing this information in the type certificate as well. If this were done, the verification task should be able to be performed in linear time (where the input size is the combined size of the bytecode program and of the certificate) and in constant space. Such a verification algorithm would be suitable for limited-resource machines such as smart cards.

3.1 Certificate checking in detail

During verification, our task is to ensure that a collection of recursive function definitions have the types specified in the certificate. To do this we examine each super-combinator definition in turn. We proceed by first *assuming* that all references to combinators in the bytecode sequence have the types given in the certificate. We then *verify* that, under the assumptions we have made, the bytecode sequence accepts the correct number of arguments of the correct types and that all paths through the control flow graph end by returning a result of the expected type.

PUSH *n*	Read the *n*th value from the stack and push it onto the top.
PUSH_ARG *n*	Read the *n*th argument to this combinator and push it onto the top of the stack.
APPLY	Take a partial application off the top of the stack and apply it to the next element on the stack. Finally, push the new application node onto the stack.
MK_CON *nm i n*	Create a data constructor node with the name *nm*, tag number *i*, and arity *n* using the top *n* elements on the stack. Push the new constructor node onto the stack.
UNPACK	Take the top element of the stack, which must point to a data constructor node, and push the elements of the constructor onto the stack.
SLIDE *n*	Take the top element off of the stack, pop the next *n* elements, and then replace the top element.
POP *n*	Pop the top *n* elements off of the stack.
ALLOC *n*	Create *n* "place-holder" nodes and push references to them on the stack.
UPDATE *n*	Overwrite the node pointed to by the $n+1$th position on the stack with the top element of the stack.
RETURN	Exit the local procedure and overwrite the current heap node with the value on top of the stack.
EVAL	Evaluate the top reference on the stack. If it is not already a value, a new stack frame will be pushed.
LOOKUP_SWITCH *l ls*	*ls* is a list of (tag-value,label) pairs. Examine the top element of the stack, which must be a data constructor. If the tag value matches any value from *ls*, jump forward to the specified label. If no value matches, jump forward to label *l*.
LABEL *l*	Mark a branch destination.
PUSH_FUNC *nm*	Push a reference to the named super-combinator onto the stack.
REWRITE_TYPE σ	Rewrite the type of the top element on the stack using the given type rewrite rule.

TABLE 4. The Stage 1 Bytecode Set

If all bytecode sequences in the module have their expected types, verification succeeds and we conclude that super-combinator definitions have the types stated in the certificate.

The core of the verification algorithm involves using abstract interpretation to fully elaborate the type of each item on the local execution stack. At the beginning of the execution of a super-combinator, the local stack is always empty. Each legal instruction in the bytecode sequence may have some effect on the local stack. For example, instructions like PUSH_ARG and MK_CON add new items to the top of the stack, while instructions such as POP and SLIDE remove items from the stack. Some instructions also have preconditions that must be true before they can execute. For example, the UNPACK instruction requires an evaluated data constructor to be on the top of the stack.

To perform verification, we first allocate an array to hold the results of the ab-

stract interpretation with one location for each program point (the spaces between bytecode instructions). The information tracked is an abstract representation of the stack, consisting of a stack of types and their status codes, the amount of heap space reserved, and the status of the arguments. A stack slot may have one of the following four status codes: Normal, Evaluated, Uninitialized, or Zapped. An argument may be either Normal or Zapped. The meanings of these status codes are explained below. The first location in the array is initialized with an empty stack, 0 reserved heap space, and with all arguments in the Normal state.

The array is then filled out by stepping down each bytecode instruction in order. If an instruction is at position i, then the information at location i in the array corresponds to the abstract state of the G-Machine before that instruction executes. Given the previous state and the instruction, we calculate values of all possible next positions and fill in those locations in the array with an appropriately modified abstract state. For most instructions, the only affected location will simply be the next location, $i+1$. Control flow instructions, however, may affect several locations in the array, and the `RETURN` instruction affects no other locations because it signals the end of the control flow for the local procedure.

Many instructions have preconditions that must be true for correct execution. As a simple example, the instruction `POP 2` requires that the stack contain at least two items. If the precondition for an instruction is violated, verification will fail.

3.2 Incorporating Type Rewrite Rules

When elaborating the stack shapes for a bytecode sequence we reuse the types from the intermediate language, which has polymorphic types, iso-recursive types, and indexed sums. In the IL, these type constructs are intimately connected to corresponding term constructs which are specifically designed to cooperate with the type system. These constructs include the type lambda and type application of System F, the `roll` and `unroll` forms, and the type annotations on data constructors. During compilation, these "non-computational" term constructs are removed from the instruction stream, following the standard type-erasure semantics. If we simply erased these constructs, however, then our verification task would almost certainly be impossible.[3]

Therefore, rather than erasing these constructs outright, we instead convert the occurrence of these type manipulating constructs into type rewrite rules. These rules form the third major component of the type certificate listed above. The rewrite rules are carefully chosen so that a rule is always safe to apply if the type matches the required form for the rule. For example, one rewrite rule replaces a polymorphic type with a specialization of that type. Other rules allow the replacement of a recursive type with its one-step unrolling, and vice versa. By using these rewrite rules, we can perform a style of type-checking that requires no unification,

[3] Without additional information, the verification task seems to require second-order unification, which is well-known to be undecidable [5].

$$\frac{t \looparrowright s}{s \mapsto t}\ (\textsc{RollType}) \qquad \frac{\Gamma \vdash_k s :: k}{(\forall \mathrm{x} :: k.\ t) \mapsto ([\mathrm{x} \mapsto s]t)}\ (\textsc{PolyAp})$$

$$\frac{s \looparrowright t}{s \mapsto t}\ (\textsc{UnrollType}) \qquad \frac{\mathrm{x} \notin \mathrm{FV}(s)}{(s \to \forall \mathrm{x} :: k.\ t) \mapsto (\forall \mathrm{x} :: k.\ s \to t)}\ (\textsc{PolyHoist})$$

$$\frac{\begin{array}{c} i' \neq i_x \qquad \text{forall } x \text{ where } 0 \leq x < m \\ \Gamma \vdash_k t' :: k \qquad t' \rhd^* \langle\!| v_0, \cdots, v_{r-1} |\!\rangle \end{array}}{\{\!| i_0 : t_0, \cdots, i_{m-1} : t_{m-1} |\!\} \mapsto \{\!| i_0 : t_0, \cdots, i' : t', \cdots, i_{m-1} : t_{m-1} |\!\}}\ (\textsc{ExpandSum})$$

In the rules PolyAp and ExpandSum, Γ refers to the type environment of the super-combinator within which the rule appears.

TABLE 5. Type rewrite rules

much like the type system for vanilla System F.[4] The rewrite rules used during verification are listed in table 5.

The rewrite rules used during verification define a subset of the type subsumption relation. In this context, we say that a type A subsumes a type B iff for every IL term x and environment Γ where $\Gamma \vdash_{ty} x :: B$, there exists an IL term y such that $\Gamma \vdash_{ty} y :: A$ and x and y have identical images under type-erasure. We call a type rewrite rule "valid" if whenever it maps a type B into a type A, then A subsumes B. Validity is a sufficient condition for a rewrite rule to be safely applied. We claim that all the rewrite rules used in the verification algorithm are valid.

In the certificate, each bytecode instruction is paired with a (possibly empty) list of rewrite rules. After the stack effects of an instruction have been calculated (but before the new state is written into the array), the rewrite rules for that instruction are applied, in the order they are listed, to the top item on the stack. The rewrite rules always transform a type into a subsuming or isomorphic type and are therefore safe to apply at any time. As with instruction effects, rewrite rules can fail if applied to invalid types. If this occurs, verification will fail.

3.3 The Role of Status Codes

Some instructions require additional preconditions that are not captured by the type system. This extra information is tracked by the status of each stack slot. The "Normal" status indicates that we have no particular knowledge about the given data item; it is the default status. The "Evaluated" status indicates that we know the data item has been reduced to weak head normal form (WHNF). The "Uninitialized" state is assigned to items on the stack that were created using the `ALLOC` instruction and will be discussed below. The "Zapped" state is used to indicate

[4] Although we call our certificates "type certificates," it would probably be more accurate to call them "typing certificates," because the certificate supplies the information necessary to reconstruct a particular type derivation.

stack items that the compiler has determined will not be used again. Zapping will also be discussed below. As mentioned above, combinator arguments may have either the "Normal" or "Zapped" states.

Data with any status except "Zapped" may be the target of the `EVAL` instruction, which is used to instruct the G-Machine to evaluate the targeted data. After `EVAL`, the status is set to evaluated. Instructions which actually examine the contents of a heap node (such as `LOOKUP_SWITCH`) require their arguments to be evaluated.

In order to improve the space-behavior of programs, the Yhc bytecode set includes instructions that can "zap" stack or argument slots to indicate that they will not be used in the future. When a stack or argument slot is zapped it is replaced with a dummy pointer; later accessing that location will generate an error. The compiler inserts zaps in certain situations where it can statically determine that data will not be accessed again in the current local procedure. The hope is that data references will be removed from the root set and that the garbage collector will therefore be able to reclaim memory sooner. Without zapping, some programs would have significantly worse space behavior than expected.

When done correctly, zap instructions are only added in places where they will not affect program execution. During verification, we check to ensure that this is true. When a location on the stack or an argument is zapped, its status is set to reflect this. If that location is accessed by later instructions, the verifier will flag this as an error.

The "Uninitialized" status exists to curb the use of a potentially dangerous instruction, `UPDATE`. The `UPDATE` instruction tells the G-Machine to *overwrite* a heap location with the data item on the top of the stack. This instruction is used in concert with `ALLOC` to implement `let`. If its use was unchecked, however, `UPDATE` could be used to break referential integrity by overwriting arbitrary heap nodes.

In order to prevent the unsafe use of `UPDATE`, we restrict it so that only data items with the status "Uninitialized" can be overwritten. The only way to create uninitialized data is to use the `ALLOC` instruction. Once a location has been overwritten, its status is set to that of the data with which it was overwritten. Thus, heap locations created by `ALLOC` have a one-shot ability to be overwritten by `UPDATE`.[5]

Finally, note that the status codes record statically-known information about the stack or argument pointer itself, and *not* information about data in the heap. If a stack slot has status "Evaluated," then it is a *pointer* which is known to point to evaluated data. There is no contradiction, for example, if one has two pointers

[5] We could additionally require that all uninitialized data be overwritten so that it does not "leak." It should be sufficient to require that no combinator return with uninitialized data on the stack and prevent uninitialized data from being evaluated or from being removed from the stack. However, we have not done this in our implementation. The Yhc runtime treats uninitialized data in a way very similar to data created using the `error` primitive. Preventing uninitialized data from escaping, therefore, does not gain anything in terms of safety, and we decided to forego the extra effort required to implement this restriction.

to the same heap data with different status codes. Also, there is no heap data associated with a "Zapped" pointer. A "Zapped" pointer is much like a null pointer in C; it does not point to valid data.

3.4 Managing Reserved Heap Space

There are a large number of Yhc instructions that require heap allocation. It is important for the runtime to be able to manage its heap efficiently. The default runtime for Yhc manages its heap using a single pointer. The pointer begins at the base of the heap and indicates the next available free location. When space is allocated, the pointer is incremented. The heap runs out of space when the next allocation would drive the heap pointer into the space occupied by the program stack. When this happens, the garbage collector is run to compact the heap. The heap pointer is then reset to the address just beyond the end of the compacted heap and execution continues. If the garbage collector cannot reclaim enough memory, the program immediately aborts.

While checking for sufficient heap space is a cheap operation, it is still not free. If each instruction that performed allocation were required to check for free space, it would incur a significant performance penalty. Instead, Yhc requires the compiler to insert NEED_HEAP instructions in the bytecode stream. A NEED_HEAP instruction reserves some amount of heap space for instructions that follow it. If the required amount is not available, the garbage collector is run to compact the heap. This allows allocating instructions to perform unchecked heap allocations.[6] The amount of space to reserve can be calculated using a simple static analysis of the bytecode.

If insufficient space is reserved, it is possible for allocations to overwrite other areas of memory. This would involve overwriting parts of execution stack because of the way Yhc's memory is laid out. While the behavior would probably be difficult to predict, it might still be possible to construct an exploit. It would certainly be possible to crash the runtime or to cause erratic behavior.

To prevent these problems, we track the amount of reserved heap space while doing verification. NEED_HEAP instructions increase the amount of reserved heap space and all instructions that perform allocation decrease it. If the amount of reserved heap minus the current height of the stack ever falls below 0, then validation fails.

3.5 Control Flow with Type Rewrite Effects

Most instructions have very straightforward abstract effects. All straight-line (non control-flow) instructions only affect the immediately following program point and most have simple effects. The PUSH instruction, for example, copies the type and status of an item further down in the stack onto the top and the ZAP_STACK and

[6] The APPLY instruction is a special case that always causes a free space check.

ZAP_ARG instructions set the status of a stack or argument location to "Zapped." Some instructions, however, have more subtle effects on the abstract state.

By far the most complicated instruction, in terms of its abstract effects, is LOOKUP_SWITCH. This instruction works by examining the top item on the stack (which must be a data constructor) and branching based on the value of the constructor's tag. This instruction is used in the translation of the `case` statement. LOOKUP_SWITCH takes two arguments: a jump target for the default branch and a list of tag-jump pairs. If the tag of the examined data item matches one of the tags in the list, then the G-Machine jumps by the appropriate amount. If no tag matches, the default branch is taken. In order for these instructions to execute correctly, the item being examined must have a sum type and it must have the Evaluated status.

The unusual thing about the LOOKUP_SWITCH instruction is that it rewrites the type of the examined data item to reflect the knowledge gained by examining the data constructor's tag. For example, if the runtime examines a data item with the type $\{|\, 0 : \mathrm{A}, 1 : \mathrm{B} \,|\}$ and discovers that the constructor tag is 0, then it can rewrite the type to $\{|\, 0 : \mathrm{A} \,|\}$ instead. Thus, whenever a tag is matched and a branch followed during verification, the sum type is *restricted* so that it only contains the variant corresponding to the observed tag. If the default branch is taken we leave the type unchanged.[7]

This type rewrite is necessary to correctly handle the bytecode sequence that arises from the translation of pattern matching. The first instruction that occurs after branching off of (a non-default arm of) a LOOKUP_SWITCH instruction is the UNPACK instruction. UNPACK takes the encapsulated pointers out of a data constructor and pushes them onto the stack. The only way we can predict the effect UNPACK will have is to know the number and types these pointers. We have this information just in case the type being unpacked is a sum type with exactly one variant, and types of this form are therefore required for UNPACK.

4 BACKGROUND AND RELATED WORK

In previous work, Leroy lays a formal foundation for secure applets and demonstrates how a static type system can be used to enforce security policy in an applet container [7]. The basic ideas laid out in that paper provide much of the motivation for this work.

Bytecode verification has been present in the Java runtime system since its inception in the mid 1990's. The Java bytecode verification algorithm is specified using prose in *The Java Virtual Machine Specification* [8]. Although early implementations of the Java verifier had some problems [2], verification has largely been successful at its task, and Java's security record is quite good. To the best of our knowledge, the basic idea of verification as a type-checking static analysis on bytecode originated with Java.

[7] We could rewrite the type to remove variants that we know are *not* in the sum; however, there does not currently seem to be any advantage to doing so, and we have opted for the simpler method.

The present work shares some similarities with previous work on typed assembly language (TAL) [10, 9]. Both start with a source language based on System F and target a lower-level instruction set with type annotations. Also, the closure conversion and hoisting phases of Morrisett's System-F-to-TAL compiler perform a translation that is quite similar to the lambda lifter used in our compiler. Work on TAL differs significantly from this work, however, because it focuses on the call-by-value evaluation model of the ML family of languages, and because it deals with a lower-level instruction set more akin to actual RISC machine instruction sets. Nonetheless, as some Haskell compilers do compile to native machine code, it may be interesting to investigate the possibility of mapping the certified G-Machine bytecode which is the target of this work down to some form of typed assembly language. This would allow a completely typed compilation pipeline all the way down to machine code.

Using extended System F as an intermediate language for compiling Haskell is not a new idea. The GHC Haskell compiler has long used an intermediate language based on F_ω [11, 13]. Indeed, the approach used in GHC was part of the inspiration for our own. Rather recently, the GHC core language was extended with support for "type-equality coercions," which form the type-theoretic basis for generalized algebraic data types (GADTs) and for associated types [17]. Although the addition of type-equality coercions comes at the cost of significant complications to the type system, there does not seem to be any fundamental reason why the IL and our certificate checking algorithms could not be similarly extended.

Starting in September 2005, the Yhc project began working on a Haskell implementation which consists of a bytecode compiler and interpreter. The Yhc project got a head start by reusing the NHC compiler. The back-end of the NHC compiler was modified to emit the newly-designed Yhc bytecode format, and a stand-alone bytecode interpreter was written to execute the compiled programs.

NHC is also a bytecode compiler [16], and its bytecode instruction set was a starting point for the development of the Yhc instruction set. However, NHC does not have a stand-alone bytecode file format. Instead, NHC emits C code which contains the bytecode stream as static data. This generated code is compiled together with the runtime using a C compiler to create the final executable.

The principal developers of the Yhc project have not addressed the issues of bytecode verification. This is because the focus of the Yhc project has not been on mobile code, the primary use-case where verification becomes compelling. Instead, the focus has largely been on portability, fast compilation, debugging tools, and fixing design problems with the original NHC compiler. Additionally, the Yhc compiler uses an untyped core language for the majority of its compilation pipeline, which would make adding the ability to produce type certificates a significant software engineering challenge. With the addition of a secure bytecode verifier, the Yhc project could easily form the basis of a trustworthy applet and distributed code execution platform.

The Yhc compiler and interpreter are both nearing readiness for real-world use; they implement the vast majority of the Haskell 98 standard as well as some

of the usual extensions. The Yhc bytecode interpreter is a stand-alone program that does not rely on the compiler proper. Thus, the Yhc bytecode interpreter makes an excellent compilation target for this project.

5 FUTURE WORK AND CONCLUSIONS

We have investigated bytecode verification for the Haskell language, specifically, the bytecode instruction set used by the Yhc Haskell compiler project. We defined an intermediate language capable of encoding a significant subset of the Haskell language. We wrote a compiler which translates our intermediate language "source" code into Yhc bytecode and an accompanying type certificate. This compiler is very basic and makes no real attempt at program optimization. However, it is careful to preserve type information all the way through the compilation pipeline so that it is available when generating the type certificate. Multiple test programs were written in the intermediate language (mostly transcribed from small Haskell programs), and were tested for correct behavior.

Finally, a bytecode verifier was written which ensures that a bytecode program is well-typed. It takes as input the executable bytecode and its accompanying certificate and outputs either a success condition code or an error. The goal is twofold: first, that any program which passes validation will be well-behaved when executed; and second, that any well-typed IL program can be correctly compiled into a bytecode program which passes validation. We subjected the verifier to basic testing by compiling the above test programs and ensuring that they passed the verifier. Additionally, several untypeable and incorrectly-typed bytecode programs were manually created to test that the verifier correctly rejects these programs.

This work represents a proof-of-concept for generating type-certified Haskell programs based on the Yhc bytecode instruction set. Issues not yet addressed include the Haskell module system (with the accompanying separate compilation issues) and the side-effectual IO monad. Also, the formal properties of this system have yet to be proven. Despite the work still to be done, we believe this work represents an important practical step along the road to a full-scale type-certifying Haskell compiler and execution system.

REFERENCES

[1] Lennart Augustsson. *Compiling lazy functional languages, part II*. PhD thesis, Department of Computer Science, Chalmers University, Sweden, 1987.

[2] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy: May 6–8, 1996, Oakland, California*, pages 190–200, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[3] Robert Dockins and Samuel Z. Guyer. Bytecode verification for haskell. Technical Report 2007-02, Department of Computer Science, Tufts University, Feburary 2007.

[4] J.-Y. Girard. Une extension de l'interprétation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. North-Holland, Amsterdam, 1971.

[5] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[6] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities. In *Proceedings of the Programming Languages Meet Program Verification Workshop*, August 2006.

[7] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[8] Xavier Leroy and François Rouaix. Security properties of typed applets. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 391–403, New York, NY, 1998.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[10] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Journal of Functional Programming*, January 2002.

[11] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[12] S. Peyton-Jones. Compiling haskell by program transformation: a report from the trenches. In *Proceedings of the European Symposium on Programming*, 1996.

[13] S. Peyton-Jones and David Lester. A modular fully-lazy lambda lifter in HASKELL. *Software - Practice and Experience*, 21(5):479–506, 1991.

[14] S. Peyton-Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Progamming*, 12, 2002.

[15] S. Peyton-Jones and Jon Salkild. The spineless tagless G-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201, New York, NY, USA, 1989. ACM Press.

[16] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[17] Niklas Röjemo. Highlights from nhc: a space-efficient haskell compiler. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 282–292, New York, NY, USA, 1995. ACM Press.

[18] Martin Sulzmann, Manuel Chakravarty, and S. Peyton-Jones. System F with type equality coersions. Submitted to TLDI, 2007.