Bytecode Verification for Haskell

Robert Dockins
Tufts University
robert.dockins@tufts.edu

Samuel Z. Guyer
Tufts University
sguyer@cs.tufts.edu

February 1, 2007

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

In this paper, we adapt the ideas of bytecode verification to the Haskell setting. Our major contribution is a working proof-of-concept implementation of a bytecode compiler and verifier. The verifier works by comparing program bytecode to a certificate which provides evidence that the bytecode is well typed. The certificate is produced by the compiler and contains type information that would otherwise be discarded during type-erasure.

Type-safety is a desirable property for programming languages because it eliminates entire classes of exploitable bugs, such as buffer overflows. Type-safety also allows programmers to reason about the security properties of their programs by reasoning within the abstraction of the source language. For example, the technique of "lightweight static capabilities" [18] encodes program invariants into the type system. Using lightweight static capabilities, one can provide APIs which will staticly ensure useful properties such as: a given list is non-empty; an array access is in bounds (without a runtime check); or that a database resource is only accessed while the database connection is open.

Type-safety is also an important ingredient of applet containers and mobile code execution platforms. Leroy showed that type-safety forms the lynch-pin of applet security and isolation systems [21]. Type-safety guarantees that programs cannot break the abstractions of the source language, and makes it possible to design security and isolation systems within the language itself. For example, we could design applet system where programs are only allowed to access the local machine by interacting with an abstract data type describing the applet's privileges. Such a type might have a list of directories where the applet is allowed to write files. If we allow the applet to somehow break the abstraction barrier and interact with its privileges descriptor in an unintended way, the applet may be able to escalate its privileges and breach the security of the host system. Type-safety ensures that this does not occur.

Although there are a number of programming languages which boast strong type-safety, Haskell is uniquely suited to be a host language for applets and mobile code execution. The primary reason for this is that

Haskell code is *purely* functional, and side-effects are only possible via a special mechanism called the IO monad. "Pure" programs (those not in the IO monad) are utterly benign: they are unable to perform any actions which cause side-effects. The only things pure computations can do are: 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. Furthermore, a program which is in the IO monad will always reflect this fact via its type. If we restrict our applet container to only execute code which in not in the IO monad, we can be confident it will be well-behaved. We can also allow limited access to side effects by designing an API which is similar to the IO monad, but restricted by a security policy.

Unfortunately, transmitting raw source code is unacceptable for mobile applications and we cannot rely on static properties of the source language for our security guarantees. Instead of source code, mobile execution systems are usually based on *bytecode*, an intermediate program representation which is easy to interpret but retains architecture portability. With a bytecode system, source programs are compiled to bytecode and the bytecode is transmitted to the remote host where it is executed. Even though the source language has static guarantees which ensure type-safety, it is possible to manually construct bytecode programs which violate the runtime's expected invariants. What we require is a system which allows us to "type-check" programs that have already been compiled to bytecode and to reject invalid programs. The process of type-checking bytecode programs is called verification.

Once a Haskell program is compiled, however, its type information is usually no longer available. Most Haskell implementations perform *type-erasure*, which removes all type information from a program before it is transformed into an executable. Our compiler is unusual in that it retains type information all the way through compilation which is used to generate a type certificate which accompanies the bytecode program. This type certificate can then used during the verification process to ensure that the bytecode program is type-safe. Once the program has been verified, the type certificate is no longer needed, and execution proceeds using only the bytecode. Our verification algorithm is lightweight and should be suitable for adaptation to limited-resource machines.

The remainder of this paper is organized as follows. Section 2 introduces our compiler intermediate language and and discusses points of note regarding its type system. Following that, section 3 briefly describes the compilation target for the compiler, the Yhc bytecode interpreter. In section 4, we covert the implementation of the IL compiler. Section 5 describes the workings of the bytecode type certificates and the verification algorithm, while section 6 gives some additional background and discusses related work. Section 7 lists possible future work and concludes.

2 The Compiler Intermediate Language

Haskell is a full-scale programming language and involves a multitude of details, many of which are only tangentally related to the matters at hand. To make the scope of this work manageable, we have focused our efforts on a core calculus (called the Intermediate Language or IL hereafter) rather than attempt to handle raw Haskell. This IL is sufficient to model the expressions and types of all of the Haskell 98 standard, given a suitable translation. However, we make no attempt at this point to model the module system or deal with the issues of separate compilation.[1]

We also do not attempt to treat side-effectual computation and the IO monad. We do this primarily because the semantics of the IO monad are considerably more difficult to model than those of the pure functional core of Haskell. However, this omission is not critical for our present aims.

The practical elements that one needs to execute the IO monad are just side-effectual primitive operations and some way to control execution order. Both these elements are easily available in the G-Machine runtime model which is the compilation target. Execution order is easily controlled using the G-Machine and we can treat side-effectual primitives in the same way as pure primitives for the purposes of verification. In fact, the verification algorithm as given in section 5 would require no changes to deal with side-effectual primitives and the IO monad.

2.1 Design Considerations

The overall strategy for a complete certifying compiler involves translating the source Haskell code to a typed intermediate language, performing lambda lifting, and then generating certified G-Machine bytecode. To suit our purposes, the intermediate language should:

- be capable of embedding all the constructs of the source language,
- be capable of representing each of the intermediate steps up to code generation,
- be as minimal as possible, and
- have a well-established meta-theory.

Clearly, these properties coexist with some tension. As the first two requirements cannot be compromised, the latter two have suffered the most. Thus the IL is more complicated than we would like and contains a few minor extensions beyond those well-studied in the literature. Nonetheless, we have tried to define the IL so that the most important meta-theoretic properties are retained, including the Church-Rosser property and type-safety. One possible avenue of future work is to formally verify these properties.

The IL is built on the base of System F_ω [10, 36]. To this base we add the following extensions:

- primitive types and functions
- term products and projections
- type products and projections
- recursive and non-recursive let
- iso-recursive types
- sum types, data constructors and case analysis
- "top-level" recursive term definitions
- "top-level" non-recursive type definitions

[1] Extending the IL to handle multiple modules should not be difficult, as the additional technical issues raised are minor.

- a message-carrying error term
- the "seq" primitive

Of these extensions, only type products are novel; the remainder are standard.[2] The full syntax of the IL is presented in table 1. Judgments relating to the type system may be found in table 5, table 6 and table 7.

2.2 IL Examples

Rather than belaboring the formal definition of the IL and its judgments, here we shall present some example IL terms that we hope will help the reader to understand the ideas behind the IL and to justify the design decisions. A more detailed discussion of the notable and unusual aspects of evaluation, typing and type erasure may be found in the following sections.

We begin with some of the simplest possible definitions in the IL and work up toward more realistic examples.

$$\begin{array}{lll} \mathit{\$bottom} & :: & \forall \textsc{a} :: \star.\ \textsc{a} \\ & \equiv & \mathit{\$bottom}; \\ \\ \mathit{\$id} & :: & \forall \textsc{a} :: \star.\ \textsc{a} \rightarrow \textsc{a} \\ & \equiv & \Lambda \textsc{a} :: \star.\ \lambda x :: \textsc{a}.\ x; \\ \\ \mathit{\$idP} & :: & \$\mathrm{P} \rightarrow \$\mathrm{P} \\ & \equiv & \mathit{\$id}\,[\$\mathrm{P}]; \end{array}$$

Here we have defined a trivially non-terminating term,*$bottom*, using nominal recursion.[3] It is assigned the the very general type $\forall \textsc{a} :: \star.\ \textsc{a}$, indicating that the term *$bottom* is a polymorphic term which can be instantiated at any type with kind $\star$. [4]

We have also defined the polymorphic identity function *$id*. Its type indicates that it is a function (at any type $\textsc{a}$ of kind $\star$) which accepts a term of type $\textsc{a}$ and returns a term of type $\textsc{a}$. In this definition, note that we use a type lambda to generate the polymorphic type. Type lambdas introduce polymorphic functions and give rise to polymorphic types. *$idP* is a specialization of *$id* to the type $P (defined later). Type application reduces with type lambdas via β-reduction so that, when reduced, *$idP* is equivalent to $\lambda x :: \$\mathrm{P}.\ x$.

Note that top-level identifiers are typographicly distinguished from bound identifiers by the leading $. Also, type variables appear in small caps. Finally, notice that the IL uses a Church-style type discipline, where binders fix the type (or kind) of bound variables. A Church-style presentation has the significant advantage that typechecking is a straightforward, top-down procedure and is decidable in many more cases than a Curry-style presentation (without type annotations on binders) [40].

We could also have defined *$bottom* by using local nominal recursion via a `reclet` statement. Similar to top-level identifiers, `reclet`-bound identifiers are distinguished by a leading % character.

$$\begin{array}{lll} \mathit{\$bottom2} & :: & \forall \textsc{a} :: \star.\ \textsc{a} \\ & \equiv & \Lambda \textsc{a} :: \star.\ \texttt{reclet}\,\{\,\%x :: \textsc{a} \equiv \%x;\ \}\ \texttt{in}\ \%x; \end{array}$$

Also noteworthy is that the `reclet` construct is recursive, and thus `reclet`-bound variables scope both over the `reclet` definitions and over the body of the `reclet`.

[2] However, the presentations of term products and sums are not standard. The subtleties relating to the two varieties of products in the IL are covered in a later section.

[3] The term "nominal recursion" is used to indicate recursion where definitions refer to themselves by name, and to distinguish it from recursion introduced by an explicit fixed-point operator.

[4] In fact, we could have assigned any valid, closed type to *$bottom*.

$k ::=$	Kinds:
$\star$	Base Kind
$k \Rightarrow k$	Arrow Kind
$\langle\!\langle\!\mid k_0, \cdots, k_{n-1} \mid\!\rangle\!\rangle$	Type-Product Kind
Π_i	Product Kind

$t ::=$	Types:
$\textsc{a}$	Type Variable
$\hat{\lambda}\textsc{a} :: k.\ t$	Type-Level Abstraction
$t\ \ t$	Type-Level Application
$\langle\!\langle\, t_0, \cdots, t_{n-1} \,\rangle\!\rangle$	Type Product
$\hat{\pi}_i\ t$	Type Projection
$t \rightarrow t$	Arrow Type
$\forall\textsc{a} :: k.\ t$	Polymorphic Type
$\mu\,\textsc{a} :: k.\ t$	Recursive Type
$@\{\!\mid p \mid\!\}$	Primitive Type
$\langle\!\mid t_0, \cdots, t_{n-1} \mid\!\rangle$	Product Type
$\{\!\mid i_0 : t_0, \cdots,$ $i_{n-1} : t_{n-1} \mid\!\}$	Sum Type

$m ::=$	Terms:
x	Term Variable
$\%x$	Let-Bound Variable
$\$x$	Top-Level Definition
$\lambda x :: t.\ m$	Abstraction
$m\ \ m$	Application
$\Lambda\textsc{x} :: k.\ m$	Type Abstraction
$m\ [t]$	Type Application
`error` $[t]$ "message"	Error
`seq` $m\ m$	Seq
$@[\,p\,]\{\,z\,\}$	Primitive
$@\{\,f \mid m_0, \cdots, m_{n-1}\,\}$	Primitive Function
$\langle\, m_0, \cdots, m_{n-1}\,\rangle$	Product
$\pi_i\ \ m$	Projection
$\&C_i\,[j_0 : t_-, \cdots, j_{n-1} : t_{n-1}]\ m$	Sum Constructor
`roll`$[t]\ m$	Recursive Roll
`unroll` m	Recursive Unroll
`let` $\{\, x\ =\ m\,;\}$ `in` m	Non-Recursive let
`reclet` $\{\,\%x_0 :: t_0 = m_0\,;$ $\cdots\,;$ $\%x_{n-1} :: t_{n-1} = m_{n-1}\,;$ $\}$ `in` m	Recursive Let
`case` m `default` m `of` $\{\quad i_0 : arm_0\,;$ $\cdots\,;$ $i_{n-1} : arm_{n-1}\,;$ $\}$	Case
`primcase`$[\,p\,]\ m$ `default` m `of` $\{\quad i_0 : arm_0\,;$ $\cdots\,;$ $i_{n-1} : arm_{n-1}\,;$ $\}$	Primitive Case

$$\textsc{IsValue}(\ \lambda x :: t.\ m\ ) \qquad (\text{V-}\textsc{Abs})$$

$$\textsc{IsValue}(\ @[\,p\,]\{\,z\,\}\ ) \qquad (\text{V-}\textsc{Prim})$$

$$\textsc{IsValue}(\ \&A_i\,[t_0, \cdots, t_{n-1}]\ m\ ) \qquad (\text{V-}\textsc{Con})$$

$$\frac{\textsc{IsValue}(\ m\ )}{\textsc{IsValue}(\ \Lambda\textsc{x} :: k.\ m\ )}\ (\text{V-}\textsc{AbsT})$$

$$\frac{\textsc{IsValue}(\ m\ )}{\textsc{IsValue}(\ \texttt{roll}[t]\ m\ )}\ (\text{V-}\textsc{Roll})$$

Table 1: Syntax of the IL and the IsValue Predicate

To make things more concrete, let us define a data type such as one might define in Haskell. A simple example of a Haskell data type is the following definition, which introduces the Peano numbers: [5]

```
data P = Z | S P
```

This declaration introduces a new type `P`. This type has two data constructors, `Z`, which corresponds to zero, and `S`, which is the successor function. Thus, `S (S Z)`, for example, corresponds to the number 2. All the usual arithmetic functions on natural numbers can be written using this data type. When translated into the IL, one obtains:

$$
\begin{aligned}
&\$\mathrm{P} && \doteq \ \mu\,\mathrm{P} :: \star.\ \{\!|\,0\ :\ \langle\!|\ \ |\!\rangle, 1\ :\ \langle\!|\,\mathrm{P}\,|\!\rangle\,|\!\};\\[1em]
&\mathit{\$Z} && ::\ \ \$\mathrm{P}\\
& && \equiv\ \ \texttt{roll}[\$\mathrm{P}]\,(\&Z_0\,[0\ :\ \langle\!|\ \ |\!\rangle, 1\ :\ \langle\!|\,\$\mathrm{P}\,|\!\rangle]\,\langle\ \rangle);\\[1em]
&\mathit{\$S} && ::\ \ \$\mathrm{P} \to \$\mathrm{P}\\
& && \equiv\ \ \lambda x :: \$\mathrm{P}.\ \texttt{roll}[\$\mathrm{P}]\,(\&S_1\,[0\ :\ \langle\!|\ \ |\!\rangle, 1\ :\ \langle\!|\,\$\mathrm{P}\,|\!\rangle]\,\langle\,x\rangle);
\end{aligned}
$$

These definitions use a lot of syntax which may be unfamiliar, so a detailed explantation follows. In the definition for $P, we see our first example of a recursive type. Recursive types are introduced by the type-variable binder μ. The body of the recursive type is a sum of products. The first component of the sum has tag 0 and is the empty product (sometimes called the unit). This first component corresponds to the `Z` constructor, which takes no arguments. The second component, with tag 1, is a unary product and corresponds to the `S` constructor, which takes a single argument of type `P`. Note that the recursion in this type definition is introduced by the μ operator, and not by nominal recursion. Also, note that the IL takes n-ary sum and product forms as primitive, rather than building them from binary sums and products, as is more typical in the literature. Finally, notice that this translation take the informal characterization of Haskell's data types as "sums of products" quite literally. The unusual empty and unary products are used to keep the translation uniform across all constructor arities.

Next, turn your attention to the translation of the `Z` constructor. The `roll` syntactic form introduces recursive data. Its type parameter indicates the recursive type to introduce. The term parameter to `roll` is a sum constructor, with the name "Z" and tag number 0. The sum constructor also takes a list of types, which fix its type. Notice this type is just the definition of $P unrolled once. The pattern where a sum type is immediately rolled into a recursive type is idiomatic of the translation of Haskell's data constructors. Finally, notice the argument to the sum constructor is just the empty product, which is natural for a data constructor of arity 0.

The `S` constructor is similar to the `Z` constructor, except that it takes a single argument. It again exhibits the sum-and-roll pattern, but the sum instead wraps a unary product containing the single argument.

Now, let's actually do something with this type we've painstakingly constructed. Here is the definition of the curried addition function on Peano numbers, first as we might define it in Haskell:

```
pAdd x y = case x of { Z -> y; S a -> pAdd a (S y) }
```

Here is the same definition translated into the IL:

[5] Actually, this definition introduces a slightly larger set than the usual Peano numbers because it admits the infinite "number" `S (S (S (···)))`, which can be usefully compared to $\aleph_0$. For our purposes, the difference is inconsequential.

$$\begin{array}{lll}
\mathit{\$pAdd} & :: & \$\mathrm{P} \to \$\mathrm{P} \to \$\mathrm{P} \\
 & \equiv & \lambda x :: \$\mathrm{P}.\ \lambda y :: \$\mathrm{P}. \\
 & & \quad \texttt{case}\ (\texttt{unroll}\ x) \\
 & & \qquad \texttt{default}\ (\texttt{error}\ [\$\mathrm{P}]\ \text{“match fail”})\ \texttt{of} \\
 & & \qquad \{\, 0 : \lambda a :: \langle\!|\ \ |\!\rangle.\ y; \\
 & & \qquad\ \ 1 : \lambda a :: \langle\!|\ \$\mathrm{P}\ |\!\rangle.\ \mathit{\$pAdd}\ \ (\pi_0\ \ a)\ \ (\mathit{\$S}\ \ y); \\
 & & \qquad \}
\end{array}$$

The most important things to note about this definition is the syntax of case analysis and use use of the `unroll` syntactic form. The `case` form has three parts: first, the case scrutinee (appears directly after the `case` keyword); second, the default branch (appears after `default`); and third, a collection of case arms (after `of` and inside the braces). The case scrutinee is the value being analyzed. It must have a sum type, and the value of its tag determines the overall value of the `case` expression. Each case arms consists of a tag number and an expression. Whenever the tag value of the scrutinee matches a case arm, the payload of the sum constructor is applied to the expression of that case arm (which must be a function of the correct type). If it occurs that the tag value of the scrutinee matches no case arm, then the default expression becomes the result of the case. Case arms are required to be ordered, so that each tag number can appear at most once on the case arm list. The `unroll` form is dual to `roll` in that they have opposite effects on the typing of an expression, and that they annihilate each other during evaluation. The idiom with `unfold` appearing as the outermost construct in a `case`'s scrutinee arises from the translation of Haskell pattern matching.

The product projection syntax is also new, and appears in the second case arm. Its operation is entirely straightforward. The projection π_i will project the ith element from a product. Product and sum components are numbered starting at 0.

Now we are going to examine the translation of one of Haskell's most ubiquitous data structures: the polymorphic list. This data structure is very much like the Peano numbers, but additionally carries a data item in each “cons” cell. The major difference in the type is that the list type is actually a type constructor: that is, a function on types. First, here is the Haskell definition: [6]

```
data List a = Nil | Cons a (List a)
```

The corresponding IL translation is:

$$\begin{array}{lll}
\$\textsc{List} & \doteq & \mu\,\mathrm{L} :: \star \Rightarrow \star.\ \hat{\lambda}\textsc{a} :: \star.\ \{\!|\, 0\ :\ \langle\!|\ \ |\!\rangle, 1\ :\ \langle\!|\, \textsc{a}, \mathrm{L}\ \ \textsc{a}\, |\!\rangle\, |\!\}; \\
 & & \\
\mathit{\$Nil} & :: & \forall\textsc{a} :: \star.\ \$\textsc{List}\ \ \textsc{a} \\
 & \equiv & \Lambda\textsc{a} :: \star.\ \texttt{roll}[\$\textsc{List}\ \ \textsc{a}]\,(\&\mathit{Nil}_0\,[0\ :\ \langle\!|\ \ |\!\rangle, 1\ :\ \langle\!|\, \textsc{a}, \$\textsc{List}\ \ \textsc{a}\, |\!\rangle]\ \langle\ \rangle); \\
 & & \\
\mathit{\$Cons} & :: & \forall\textsc{a} :: \star.\ \textsc{a} \to \$\textsc{List}\ \ \textsc{a} \to \$\textsc{List}\ \ \textsc{a} \\
 & \equiv & \Lambda\textsc{a} :: \star.\ \lambda x :: \textsc{a}.\ \lambda xs :: \$\textsc{List}\ \textsc{a}.\ \texttt{roll}[\$\textsc{List}\ \ \textsc{a}]\,(\&\mathit{Cons}_1\,[0\ :\ \langle\!|\ |\!\rangle, 1\ :\ \langle\!|\, \textsc{a}, \$\textsc{List}\ \ \textsc{a}\, |\!\rangle]\ \langle\, x, xs\,\rangle);
\end{array}$$

The main points of interest in the definition of the $\textsc{List} type are the presence of a type-level lambda [7] and the higher-kinded type operator L.

The astute reader may wonder why the μ binder appears outside the type-level lambda in the definition of $\textsc{List}. As it happens, the $\textsc{List} type *could* be defined alternately as follows:

$$\begin{array}{lll}
\$\textsc{List}2 & \doteq & \hat{\lambda}\textsc{a} :: \star.\ \mu\,\mathrm{L} :: \star.\ \{\!|\, 0\ :\ \langle\!|\ \ |\!\rangle, 1\ :\ \langle\!|\, \textsc{a}, \mathrm{L}\, |\!\rangle\, |\!\};
\end{array}$$

[6] Although the list type actually has special syntax in Haskell, it is purely cosmetic. The list datatype could be defined exactly as given here.

[7] Type-level lambdas are typographicly distinguished from term-level lambdas by the carat symbol.

This definition is simpler and therefore, it could be argued, better. However, type definitions of this form (with the type-level lambdas outermost) only work for the so-called “regular” data types: those where the type parameters of the data type do not vary. The “nested” data types popularized by Richard Bird require the more general formation [2]. To maintain consistency, all data types are translated in the same way, which gives rise to the type shown for $List. Nested data types are useful because they can be used to allow the type system to maintain non-trivial data structure invariants [3, 30].

The definition of mutually-recursive data types requires some care. Because the type system does not use nominal recursion, all recursion must be introduced by the μ binder. In order to make mutually-recursive definitions simpler, type products have been introduced into the IL. Type products are like term products simply lifted to the level of types, and they have a corresponding kinding rule. Note that type products should not be confused with the product type (the type of term products), which appears similar at first glance.[8]

To illustrate the translation of mutually-recursive datatypes, consider the following Haskell data type definitions:

```
data A = MkA B
data B = MkB A
```

By using a strongly-connected-component analysis, the Haskell front-end can discover that these type definitions are mutually recursive. It will then translate these types and their constructors as follows:

$$\begin{array}{lll}
\$\mathrm{AB} & \doteq & \mu\,\mathrm{AB} :: \langle\!\langle\!| \star, \star |\!\rangle\!\rangle.\ \langle\!\langle\, \{\!|\, 0\ :\ \langle\!|\, \hat{\pi}_1\ \mathrm{AB} \,|\!\rangle \,|\!\}, \{\!|\, 0\ :\ \langle\!|\, \hat{\pi}_0\ \mathrm{AB} \,|\!\rangle \,|\!\} \,\rangle\!\rangle; \\
\\
\$\mathrm{A} & \doteq & \hat{\pi}_0\ \$\mathrm{AB}; \\
\\
\$\mathrm{B} & \doteq & \hat{\pi}_1\ \$\mathrm{AB}; \\
\\
\mathit{\$MkA} & :: & \$\mathrm{B} \rightarrow \$\mathrm{A} \\
 & \equiv & \lambda x :: \$\mathrm{B}.\ \mathtt{roll}[\$\mathrm{A}]\,(\&MkA_0\,[0\ :\ \langle\!|\, \$\mathrm{B} \,|\!\rangle]\ \langle\, x \rangle); \\
\\
\mathit{\$MkB} & :: & \$\mathrm{A} \rightarrow \$\mathrm{B} \\
 & \equiv & \lambda x :: \$\mathrm{A}.\ \mathtt{roll}[\$\mathrm{B}]\,(\&MkB_0\,[0\ :\ \langle\!|\, \$\mathrm{A} \,|\!\rangle]\ \langle\, x \rangle);
\end{array}$$

The important thing to notice about these definitions is that the type $AB is defined by recursion over a type product. The individual components of the tuple represent the various types being defined by mutual recursion. While the type products are not strictly necessary to define mutually recursive types, the alternate translation without them is significantly more complicated.

2.3 Formal Presentation of the IL

The compiler intermediate language is based on the well-known higher-order polymorphic lambda calculus, also known as system F_ω. In basic F_ω, there are three syntactic categories: terms, types, and kinds. The IL retains these syntactic categories and extends them with additional constructs needed to model the source language.

Before discussing the features of the IL in detail, it is important to state the things with are specificly *not* modeled by the IL. The biggest feature of Haskell which is not modeled by the IL is side-effectual computation. In Haskell, side effects are captured in the abstract IO monad and its associated side-effectual primitive operations. The IL, however, only models primitive operations which perform no side effects. This

[8] See section 2.3.1 for a discussion of the differences.

decision is a deliberate simplification. It makes the presentation of the IL shorter than it would otherwise be and makes it possible to directly specify and implement the semantics of the IL as a call-by-name term reduction. The task of finding a good way to extend the IL to model Haskell's IO monad is left for future work. [9]

Another major features of Haskell that is missing is the set of primitive types: Int, Integer, Float, Double and Char. Instead, the IL is parameterized by an arbitrary set of primitives, primitive types and primitive function symbols. This system is sufficient to encode each of the primitive types in Haskell. It also reduces the number of evaluation and typing rules which must be dedicated to their treatment while at the same time allowing a straightforward way to treat the many primitive types that can be introduced as language extensions. The primitive system is covered in more detail below.

However, this does produce one odd discrepancy. In Haskell, the `error` form takes an arbitrary string argument which is printed to the console if an unrecoverable error occurs. To faithfully represent this primitive in the IL, we would have to explicitly treat both characters and lists.[10] Instead, we have added an `error` form to the IL which takes a constant string as part of it's syntax. Thus the Haskell translation to the IL can only handle compile-time constant strings. This is unfortunate, but the loss of expression is acceptable in the face of the simplification it allows. Extending the IL to fully handle this case would be straightforward.

Finally, the Haskell type-class system is not represented in the IL. Most Haskell implementations handle type-classes by using the dictionary-passing mechanism [1], whereby type-classes are translated into tuples of functions (called dictionaries) and functions with class constraints are given additional arguments to accept these dictionaries. Dictionary-passing has the advantage of being simple to understand and allows one to translate into lower-level languages which then do not need to deal with the complexities of overloading and type-class resolution. As such, the IL assumes a Haskell front-end which performs the dictionary-passing transformation, which significantly simplifies the type system.[11]

For reasons of space, we cannot go over each of the syntax constructs and various judgments in detail. Thus, we intend only to cover unusual features or cases where standard features interact in non-obvious ways.

2.3.1 Products of a Deranged Mind

On of the most confusing aspect of the IL is the fact that there are seven different syntactic forms relating to two different sorts of products. Even worse, neither of these products is precisely the same as the tuple forms available in Haskell! Finally, despite the goal of building the IL around constructs well-studied in the literature, we have chosen to take n-ary products as basic rather than the more usual binary products. In what follows we shall attempt to explain the various constructs and to defend these design decisions.

2.3.2 Term-Level Products

The term product is written using single angle brackets: $\langle\ \rangle$. Each term product has 0 or more components (where components are typographicly separated by commas) and each component has a projection, written π_i for the ith component. The type of term products mirrors the term structure, in that it is also an n-ary construct and the components of the product type correspond to the components of the term product. Product types are written using angle brackets with a vertical line: $\langle\!|\quad|\!\rangle$. Unlike the most straightforward

[9] As mentioned above, we do not feel that this omission is critical for the verification process.

[10] Strings in Haskell are simply lists of characters.

[11] Although most Haskell implementations use a dictionary-passing translation, it is not required. An alternative approach is to use an intermediate language based on Pure Type Systems (PTS) [34]. Using PTS, type classes can be implemented using a type-case construct which examines the type of the argument. The JHC Haskell implementation takes this approach. The major disadvantage of using the PTS and type-case approach is that it requires a whole-program analysis, and thus is difficult to reconcile with separate compilation.

extension of F_ω with products, the kind of IL products is distinguished from the base kind, and is written Π_n, where n is the arity of the product.

The term product is unlifted and differs from the products found in Haskell, which are lifted.[12]

The unlifted product is found in the IL for two main purposes. The first involves the translation of pattern matching, and the second has to do with primitive operations.

Before the translation to IL, Haskell pattern matching is desugared into a simpler form. In this form all instances of pattern matching are rewritten into (possibly nested) uses of the `case` statement, where each pattern is in one of two forms. The first form is that of simple, linear patterns: those of the form $C\ x_0\ \cdots\ x_{n-1}$ where C is a data constructor with arity n and all the x_i are distinct pattern variables. The second form is the wildcard pattern (written as a single underscore in Haskell), which matches any input.

Once in this form, Haskell `case` statements can be translated directly into IL `case` statements. Data constructor patterns are translated into IL case arms. The tag number of the data constructor replaces its name, and the pattern variables are replaced by a lambda which binds a product type with the appropriate arity and types. Occurrences of the pattern variables in the body of the case arm are replaced by projections from the product. The wildcard pattern (if present), becomes the default branch of the IL case form. If there is no wildcard pattern, then the translation will insert an `error` term in the default branch. Table 2 gives an example of a pattern matching translation.

The other major use of products has to do with primitive operations. There are some cases where primitive operations are best defined to return multiple results. For example, take the case of integer division. It is possible to implement integer division such that it calculates both the quotient and remainder simultaneously. If one requires both the remainder and the quotient, it is probably more efficient to calculate both with one operation than to calculate each separately.

In order to accommodate primitives which return multiple results, we have arranged for all primitives to be functions from a tuple of arguments to a tuple of results. This means that, in the IL, every primitive function construct reduces to a tuple containing the results. In the common case where there is only one result, the tuple will be the unary tuple with only one component. Product projection is used to access the results.

The unusual kinding rule for term products deserves special mention (see table 5). Notice that the type rule K-Prod ensures that each type component of a product type has kind $\star$. The reason for this is an implementation issue. At runtime, all items of the types of kind $\star$ are represented by nodes in the heap that might be either unevaluated thunks, a partial applications, or constructor nodes. Each such item can be referenced by using a single word (heap pointer) which gives the address of the item in the heap. Products, however, are unlifted, and thus are never represented by heap thunks. Because of this, we can construct a product by simply allocating enough words to hold its components either in the heap (as part of another object) or on the computation stack. If we force a product to contain only types of kind $\star$ then we can staticly calculate the amount of space required to represent a product just by knowing its kind. In particular, we can preallocate space even for polymorphic products. Furthermore, product projections can be very easily compiled into static offset calculations from the base of the product, which plays nicely into the stack-machine model which is the compilation target.

As a happy accident, the $\star$ symbol resembles the C pointer operator and can be considered mnemonic for a data item represented by a pointer to a heap object. The product kind Π_i can be viewed as an array of i pointers to heap objects.

[12] The difference between lifted and unlifted products is best explained in the context of domain theory. An unlifted product is one where $\bot = (\bot, \bot)$, and a lifted product has an "extra" bottom, so that $\bot \neq (\bot, \bot)$. In Haskell, products (AKA tuples) are just special syntax for a sum-of-products with exactly one sum component. In the IL this is made explicit by translating Haskell tuples as single-component sums of *unlifted* products. Thus the sum provides the required "extra bottom."

The original Haskell definition

```
data Either a b = Left a | Right b

filterLeft :: List (Either a b) -> List a
filterLeft Nil = Nil
filterLeft (Cons (Left a) xs) = Cons a (filterLeft xs)
filterLeft (Cons _        xs) = filterLeft xs
```

The desugared Haskell definition

```
filterLeft :: List (Either a b) -> List a
filterLeft =
    \x -> case x of
           Nil -> Nil
           Cons p0 xs ->
              case p0 of
                 Left a -> Cons a (filterLeft xs)
                 _      -> filterLeft xs
```

The translation into IL

$$
\begin{array}{lll}
\$\textsc{Either} & \doteq & \mu\,\textsc{Either} :: \star \Rightarrow \star \Rightarrow \star.\ \hat{\lambda}\textsc{a} :: \star.\ \hat{\lambda}\textsc{b} :: \star.\ \{\!|\, 0\ :\ \langle\!|\ \textsc{a}\ |\!\rangle, 1\ :\ \langle\!|\ \textsc{b}\ |\!\rangle\ |\!\};\\
 & & \\
\mathit{\$Left} & :: & \forall\textsc{a} :: \star.\ \forall\textsc{b} :: \star.\ \textsc{a} \rightarrow (\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b}\\
 & \equiv & \Lambda\textsc{a} :: \star.\ \Lambda\textsc{b} :: \star.\ \lambda x :: \textsc{a}.\ \texttt{roll}[(\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b}]\,(\&Left_0\,[0\ :\ \langle\!|\ \textsc{a}\ |\!\rangle, 1\ :\ \langle\!|\ \textsc{b}\ |\!\rangle]\ \langle\, x\rangle);\\
 & & \\
\mathit{\$Right} & :: & \forall\textsc{a} :: \star.\ \forall\textsc{b} :: \star.\ \textsc{b} \rightarrow (\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b}\\
 & \equiv & \Lambda\textsc{a} :: \star.\ \Lambda\textsc{b} :: \star.\ \lambda x :: \textsc{b}.\ \texttt{roll}[(\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b}]\,(\&Right_1\,[0\ :\ \langle\!|\ \textsc{a}\ |\!\rangle, 1\ :\ \langle\!|\ \textsc{b}\ |\!\rangle]\ \langle\, x\rangle);\\
 & & \\
\mathit{\$filterLeft} & :: & \forall\textsc{a} :: \star.\ \forall\textsc{b} :: \star.\ \$\textsc{List}\ \ ((\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b}) \rightarrow \$\textsc{List}\ \ \textsc{a}\\
 & \equiv & \Lambda\textsc{a} :: \star.\ \Lambda\textsc{b} :: \star.\ \lambda x :: \$\textsc{List}\ \ ((\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b}).\\
 & & \quad \texttt{case}\ (\texttt{unroll}\ x)\\
 & & \quad\quad \texttt{default}\ (\texttt{error}\ [\$\textsc{List}\ \ \textsc{a}]\ \text{“Match Fail”})\ \texttt{of}\\
 & & \quad\quad \{\,0 : \lambda p :: \langle\!|\ \ |\!\rangle.\ \mathit{\$Nil}\,[\textsc{a}];\\
 & & \quad\quad\ \ 1 : \lambda p :: \langle\!|\ (\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b}, \$\textsc{List}\ \ ((\$\textsc{Either}\ \ \textsc{a})\ \ \textsc{b})\ |\!\rangle.\\
 & & \quad\quad\quad\quad \texttt{case}\ (\texttt{unroll}\ (\pi_0\ \ p))\\
 & & \quad\quad\quad\quad\quad \texttt{default}\ (\mathit{\$filterLeft}\,[\textsc{a}]\ [\textsc{b}]\ \ (\pi_1\ \ p))\ \texttt{of}\\
 & & \quad\quad\quad\quad\quad \{\,0 : \lambda q :: \langle\!|\ \textsc{a}\ |\!\rangle.\ \mathit{\$Cons}\,[\textsc{a}]\ \ (\pi_0\ \ q)\ \ (\mathit{\$filterLeft}\,[\textsc{a}]\ [\textsc{b}]\ \ (\pi_1\ \ p));\\
 & & \quad\quad\quad\quad\quad \}\\
 & & \quad\quad \}
\end{array}
$$

Table 2: Example pattern matching translation

2.3.3 Type-Level Products

In F_ω, the types form a term system unto themselves, with the kinds as their type system. In particular, it is the simply-typed lambda calculus enriched with a small number of primitive forms. Type-level products are simply another enrichment to this type-level lambda calculus. Type-level products are written using double angle brackets: $\langle\!\langle\ \ \rangle\!\rangle$. Like term-products, type-products are unlifted[13] and are eliminated by using a projection operator: $\hat{\pi}_i$. Type-level products also have a kinding rule that is completely analogous to the type rule for term-level products. The kind of type-level products is written using double angle brackets with a vertical line: $\langle\!\langle|\ \ |\rangle\!\rangle$.

As we mentioned in section 2.2, type-products exist primarily as a technical device to simplify the definition of mutually-recursive data types. Data types that are defined by mutual recursion in the source language are translated into the fixpoint of a type product. The individual components of the product are then projected out to obtain the desired data types. Because mutually-recursive datatypes are defined this way, type-level products interact with recursive type unrolling in a way that may not be obvious at first. When attempting to unroll a recursive type, one must “look through” type-level projection. The rule UR-Proj from table 5 captures this notion.

2.3.4 Recursive Data Types

In Haskell recursive data types are introduced by using the `data` or `newtype` keywords. The `data` keyword introduces a new named sum-of-products data type together with its constructors. The `newtype` keyword also introduces a new named type, but the new type always has exactly one constructor and is unlifted. Additionally, the `newtype` keyword is guaranteed by the language definition to have the same runtime representation as the wrapped type [32].

Both keywords introduce recursive types because the introduced type name can appear in the definition. Furthermore, these keywords are the only way to define recursive types in Haskell.

Because Haskell does not grant the ability to define arbitrary recursive types, we can use an iso-recursive system to model the types rather than the more technically challenging equi-recursive model. In an equi-recursive model, a recursive type and its one-step unrolling are considered *equivalent* and the typechecker is expected to discover this equivalence wherever necessary. In an iso-recursive model, a recursive type and its one-step unrolling are merely *isomorphic* and the isomorphism is witnessed by injection and projection functions which are called “roll” and “unroll,” respectively (or sometimes “in” and “out”). The metatheory and implementation of iso-recursive type systems are considerably simpler than equi-recursive systems.

For these reasons, we have chosen to implement an iso-recursive system for the IL. The type isomorphism is witnessed by the `roll` and `unroll` syntactic forms, which exist for this purpose. Their treatment in the IL is standard except that we have extended the contexts in which a type can be unrolled.

The main point of interest concerning recursive types in the IL is the translation from Haskell. A number of examples of the translation of `data` declarations where presented in section 2.2, so we shall only touch on the main ideas. The basic idea is that Haskell data types are translated into a definition of a recursive sum-of-products. Each data constructor is translated into a function which takes some number (possibly 0) of arguments, wraps those arguments in a product, wraps that product in a sum constructor, and then wraps the sum constructor in `roll`. Conversely, pattern matching on a data type is translated into an IL case statement where unroll is applied to the case scrutinee.

The case for `newtype` is simpler, but also more subtle. As with `data`, each `newtype` declaration is turned into a recursive type. However, now there is no sum-of-products, only the body of the `newtype`. The `newtype`

[13] Actually, all the constructs of the type-system-considered-as-a-term-system are unlifted because evaluation is strongly normalizing.

$$\frac{\$x :: t \equiv m}{\Delta \vdash_{e} \ \$x \rhd m} \qquad \text{(E-\textsc{Def})}$$

$$\frac{(\%x, m) \in \Delta}{\Delta \vdash_{e} \ \%x \rhd m} \qquad \text{(E-\textsc{LetVar})}$$

$$\Delta \vdash_{e} \ (\Lambda \mathrm{X} :: k.\ m)\ [t] \ \rhd [\mathrm{X} \mapsto t]m \ \text{(E-\textsc{TyBeta})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\Delta \vdash_{e} \ (\Lambda \mathrm{X} :: k.\ l) \rhd (\Lambda \mathrm{X} :: k.\ m)} \qquad \text{(E-\textsc{TyLam})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\Delta \vdash_{e} \ l\ [t] \rhd m\ [t]} \qquad \text{(E-\textsc{TyApp})}$$

$$\Delta \vdash_{e} \ (\lambda x :: t.\ m)\ \ l \rhd [x \mapsto l]m \qquad \text{(E-\textsc{Beta})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\Delta \vdash_{e} \ (l\ \ j) \rhd (m\ \ j)} \qquad \text{(E-\textsc{App})}$$

$$\Delta \vdash_{e} \ \texttt{error}\ [t]\ \mathrm{msg} \rhd \texttt{error}\ [t]\ \mathrm{msg} \ \text{(E-\textsc{Error})}$$

$$\Delta \vdash_{e} \ (\texttt{let}\ \{\ x \ = \ l\ ;\ \}\ \texttt{in}\ m) \rhd [x \mapsto l]m \qquad \text{(E-\textsc{Let})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\Delta \vdash_{e} \ \pi_i \ l \rhd \pi_i \ m} \qquad \text{(E-\textsc{Proj})}$$

$$\Delta \vdash_{e} \ \pi_i \ \langle\, m_0, \cdots, m_{n-1} \,\rangle \rhd m_i \qquad \text{(E-\textsc{ProjProd})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\Delta \vdash_{e} \ \texttt{seq}\, l\ j \rhd \texttt{seq}\, m\ j} \qquad \text{(E-\textsc{Seq1})}$$

$$\frac{\textsc{IsValue}(\ m\ )}{\Delta \vdash_{e} \ \texttt{seq}\, m\ j \rhd j} \qquad \text{(E-\textsc{Seq2})}$$

$$\Delta \vdash_{e} \ \texttt{unroll}\,(\texttt{roll}[t]\ m) \rhd m \quad \text{(E-\textsc{UnrollRoll})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\Delta \vdash_{e} \ \texttt{unroll}\, l \rhd \texttt{unroll}\, m} \qquad \text{(E-\textsc{Unroll})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\Delta \vdash_{e} \ \texttt{roll}[t]\ l \rhd \texttt{roll}[t]\ m} \qquad \text{(E-\textsc{Roll})}$$

$$\frac{\begin{array}{c} \vdash_{\mathit{prim}} f :: (p0, \cdots, p_{n-1}) \Rrightarrow (q0, \cdots, q_{r-1}) \\ \delta_f(x_0, \cdots, x_{n-1}) = (z_0, \cdots, z_{r-1}) \end{array}}{\begin{array}{c} \Delta \vdash_{e} \ @\{\ f \mid @[\,p_0\,]\{\ x_0\ \}, \cdots, @[\,p_{n-1}\,]\{\ x_{n-1}\ \}\ \} \rhd \\ \langle\ @[\ q_0\ ]\{\ z_0\ \}, \cdots, @[\ q_{r-1}\ ]\{\ z_{r-1}\ \}\ \rangle \end{array}} \qquad \text{(E-\textsc{PrimFunc1})}$$

$$\frac{\Delta \vdash_{e} \ l \rhd m}{\begin{array}{c} \Delta \vdash_{e} \ @\{\ f \mid @[\,p_0\,]\{\ x_0\ \}, \cdots, @[\,p_{n-1}\,]\{\ x_{n-1}\ \}, l, m_0, \cdots, m_{r-1}\ \} \rhd \\ @\{\ f \mid @[\,p_0\,]\{\ x_0\ \}, \cdots, @[\,p_{n-1}\,]\{\ x_{n-1}\ \}, m, m_0, \cdots, m_{r-1}\ \} \end{array}} \qquad \text{(E-\textsc{PrimFunc2})}$$

Evaluation is defined by a small-step reduction relation. Taking the reflexive, transitive closure of the small-step semantics gives the usual reduction relation. In the definition of evaluation, Δ refers to an environment which binds reclet variables to their definitions. The environment is enriched when evaluating under reclet, and is initially empty. Evaluation occurs in the context of a particular module, and the rule E-Def refers to definitions occuring in the module.

Table 3: Evaluation Rules for the IL, Part 1

$$\mathit{defs} \vdash_{\mathit{reclet}} x \hookrightarrow x \qquad (\text{PL-Var})$$

$$\mathit{defs} \vdash_{\mathit{reclet}} @[\,p\,]\{\,x\,\} \hookrightarrow @[\,p\,]\{\,x\,\} \qquad (\text{PL-Prim})$$

$$\mathit{defs} \vdash_{\mathit{reclet}} \$x \hookrightarrow \$x \qquad (\text{PL-Def})$$

$$\begin{array}{c} \mathit{defs} \vdash_{\mathit{reclet}} \lambda x :: t.\ m \hookrightarrow \\ \lambda x :: t.\ \texttt{reclet}\,\{\,defs\}\ \texttt{in}\, m \end{array} \qquad (\text{PL-Abs})$$

$$\begin{array}{c} \mathit{defs} \vdash_{\mathit{reclet}} \Lambda \mathrm{X} :: k.\ m \hookrightarrow \\ \Lambda \mathrm{X} :: k.\ \texttt{reclet}\,\{\,defs\}\ \texttt{in}\, m \end{array} \qquad (\text{PL-AbsT})$$

$$\begin{array}{c} \mathit{defs} \vdash_{\mathit{reclet}} \texttt{roll}[t]\ m \hookrightarrow \\ \texttt{roll}[t]\ \texttt{reclet}\,\{\,defs\}\ \texttt{in}\, m \end{array} \qquad (\text{PL-Roll})$$

$$\begin{array}{c} \mathit{defs} \vdash_{\mathit{reclet}} \langle\, m_0, \cdots, m_{n-1}\,\rangle \hookrightarrow \\ \langle\, \texttt{reclet}\,\{\,defs\}\ \texttt{in}\, m_0, \cdots, \texttt{reclet}\,\{\,defs\}\ \texttt{in}\, m_{n-1}\,\rangle \end{array} \qquad (\text{PL-Prod})$$

$$\begin{array}{c} \mathit{defs} \vdash_{\mathit{reclet}} \&A_i\,[t_0, \cdots, t_{n-1}]\ m \hookrightarrow \\ \&A_i\,[t_0, \cdots, t_{n-1}]\ \texttt{reclet}\,\{\,defs\}\ \texttt{in}\, m \end{array} \qquad (\text{PL-Con})$$

$$\frac{\mathit{defs} \vdash_{\mathit{reclet}} l \hookrightarrow m}{\Delta \vdash_{e}\ \texttt{reclet}\,\{\,defs\}\ \texttt{in}\, l \rhd m} \qquad (\text{E-PushRecLet})$$

$$\frac{(\%x_0, m_0), \cdots, (\%x_{n-1}, m_{n-1}), \Delta \vdash_{e}\ l \rhd m}{\begin{array}{c} \Delta \vdash_{e}\ \texttt{reclet}\,\{\,\%x_0 :: t_0 = m_0; \cdots; \%x_{n-1} :: t_{n-1} = m_{n-1}\}\ \texttt{in}\, l \rhd \\ \texttt{reclet}\,\{\,\%x_0 :: t_0 = m_0; \cdots; \%x_{n-1} :: t_{n-1} = m_{n-1}\}\ \texttt{in}\, m \end{array}} \qquad (\text{E-RecLet})$$

$$\frac{\Delta \vdash_{e}\ l \rhd m}{\begin{array}{c} \Delta \vdash_{e}\ \texttt{case}\ l\ \texttt{default}\ d\ \texttt{of}\ \{\,j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\,\} \rhd \\ \texttt{case}\ m\ \texttt{default}\ d\ \texttt{of}\ \{\,j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\,\} \end{array}} \qquad (\text{E-Case})$$

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

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

$$\frac{\Delta \vdash_{e}\ l \rhd m}{\begin{array}{c} \Delta \vdash_{e}\ \texttt{primcase}[\,p\,]\ l\ \texttt{default}\ d\ \texttt{of}\ \{\,j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\,\} \rhd \\ \texttt{primcase}[\,p\,]\ m\ \texttt{default}\ d\ \texttt{of}\ \{\,j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\,\} \end{array}} \qquad (\text{E-PrimCase})$$

$$\frac{j_x = \gamma_p(z)}{\Delta \vdash_{e}\ \texttt{primcase}[\,p\,]\ (@[\,p\,]\{\,z\,\})\ \texttt{default}\ d\ \texttt{of}\ \{\,j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\,\} \rhd a_x} \qquad (\text{E-PrimCaseArm})$$

$$\frac{\nexists x\ j_x = \gamma_p(z)}{\Delta \vdash_{e}\ \texttt{primcase}[\,p\,]\ (@[\,p\,]\{\,z\,\})\ \texttt{default}\ d\ \texttt{of}\ \{\,j_0 : a_0; \cdots; j_{n-1} : a_{n-1}\,\} \rhd d} \qquad (\text{E-PrimCaseDefault})$$

Table 4: Evaluation Rules for the IL, Part 2

$$\textsc{KGood}(\ \star\ ) \qquad \text{(}\textsc{KGood-Star}\text{)}$$

$$\textsc{KGood}(\ \Pi_i\ ) \qquad \text{(}\textsc{KGood-Prod}\text{)}$$

$$(\mu\,\mathrm{X} :: k.\ t) \looparrowright [\mathrm{X} \mapsto \mu\,\mathrm{X} :: k.\ t]t \qquad \text{(UR-}\textsc{Mu}\text{)}$$

$$\frac{s \looparrowright t}{s\ \ u \looparrowright t\ \ u} \qquad \text{(UR-}\textsc{App}\text{)}$$

$$\frac{s \looparrowright t}{\hat{\pi}_i\ s \looparrowright \hat{\pi}_i\ t} \qquad \text{(UR-}\textsc{Proj}\text{)}$$

$$\frac{\$\mathrm{X} \doteq t}{\$\mathrm{X}\ \triangleright\ t} \qquad \text{(}\textsc{TyE-Def}\text{)}$$

$$(\hat{\lambda}x :: k.\ t)\ \ s\ \triangleright\ [x \mapsto s]t \qquad \text{(}\textsc{TyE-Beta}\text{)}$$

$$\hat{\pi}_i\ \langle\!\langle\, t_0, \cdots, t_{n-1}\,\rangle\!\rangle\ \triangleright\ t_i \qquad \text{(}\textsc{TyE-ProdProj}\text{)}$$

$$\frac{x\ \triangleright\ y}{x\ \ z\ \triangleright\ y\ \ z} \qquad \text{(}\textsc{TyE-App}\text{)}$$

$$\frac{x\ \triangleright\ y}{\hat{\pi}_i\ x\ \triangleright\ \hat{\pi}_i\ y} \qquad \text{(}\textsc{TyE-Proj}\text{)}$$

$$\Gamma \vdash_{k} @\{\!|\ p\ |\!\} :: \star \qquad \text{(K-}\textsc{Prim}\text{)}$$

$$\frac{(x, k) \in \Gamma}{\Gamma \vdash_{k} x :: k} \qquad \text{(K-}\textsc{Var}\text{)}$$

$$\frac{(x, k), \Gamma \vdash_{k} t :: j \qquad \textsc{KGood}(\ j\ )}{\Gamma \vdash_{k} (\forall x :: k.\ t) :: j} \qquad \text{(K-}\textsc{All}\text{)}$$

$$\frac{(x, k), \Gamma \vdash_{k} t :: j}{\Gamma \vdash_{k} (\hat{\lambda}x :: k.\ t) :: k \Rightarrow j} \qquad \text{(K-}\textsc{Lam}\text{)}$$

$$\frac{\Gamma \vdash_{k} t_1 :: k_1 \Rightarrow k_2 \qquad \Gamma \vdash_{k} t_2 :: k_1}{\Gamma \vdash_{k} t_1\ \ t_2 :: k_2} \qquad \text{(K-}\textsc{App}\text{)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_{k} t_1 :: k_1 & \textsc{KGood}(\ k_1\ ) \\ \Gamma \vdash_{k} t_2 :: k_2 & \textsc{KGood}(\ k_2\ )\end{array}}{\Gamma \vdash_{k} t_1 \rightarrow t_2 :: \star} \qquad \text{(K-}\textsc{Arr}\text{)}$$

$$\frac{(x, k), \Gamma \vdash_{k} t :: j}{\Gamma \vdash_{k} (\mu\, x :: k.\ t) :: j} \qquad \text{(K-}\textsc{Mu}\text{)}$$

$$\frac{\Gamma \vdash_{k} t :: \langle\!\langle\!|\ k_0, \cdots, k_{n-1}\ |\!\rangle\!\rangle}{\Gamma \vdash_{k} \hat{\pi}_i\ t :: k_i} \qquad \text{(K-}\textsc{Proj}\text{)}$$

$$\frac{\Gamma \vdash_{k} t_i :: k_i \qquad \text{for all } 0 \leq i < n}{\Gamma \vdash_{k} \langle\!\langle\, t_0, \cdots, t_{n-1}\,\rangle\!\rangle :: \langle\!\langle\!|\ k_0, \cdots, k_{n-1}\ |\!\rangle\!\rangle} \qquad \text{(K-}\textsc{TyProd}\text{)}$$

$$\frac{\Gamma \vdash_{k} t_i :: \star \qquad \text{for all } 0 \leq i < n}{\Gamma \vdash_{k} \langle\!|\ t_0, \cdots, t_{n-1}\ |\!\rangle :: \Pi_n} \qquad \text{(K-}\textsc{Prod}\text{)}$$

$$\frac{\begin{array}{c} t_i\ \triangleright\ \langle\!|\ v_0, \cdots, v_{r-1}\ |\!\rangle \\ \Gamma \vdash_{k} t_i :: k_i \qquad \text{for all } 0 \leq i < n\end{array}}{\Gamma \vdash_{k} \{\!|\ j_0 : t_0, \cdots, j_{n-1} : t_{n-1}\ |\!\} :: \star} \qquad \text{(K-}\textsc{Sum}\text{)}$$

Similar to term evaluation, type evaluation is defined by a small-step semantics. Taking the reflexive, transitive closure of the above relation gives the usual reduction relation. In the kinding and typing relations, Γ refers to an environment which binds type variables to their kinds and term variables to their types. As with the E-Def rule, the TyE-Def rule refers to definitions in scope in the current module.

Table 5: Kinding, Type Evaluation, and Unrolling Rules for the IL

constructor translates simply into `roll` and pattern matching on a `newtype` becomes an application of `unroll`. The reason we say that this is subtle is because `newtype` is rarely used for its ability to create recursive types. Most `newtype` (and many `data`) declarations do not mention the name of the type being defined. In this case the declaration is trivially recursive. Defining such "non-recursive" types in terms of explicitly recursive types is a little counter-intuitive, but it means that the translation from Haskell is simple and elegant. The rule is "all `data` and `newtype` declarations create recursive types," even if the type is only trivially recursive.

One final point about recursive types bears mentioning. Haskell's type system contains a mixture of structural and nominative elements, whereas the type system for the IL is purely structural. The two main features that define a nominative type system are:

- two types with the same name are guaranteed to have the same structure, and
- two types with the same structure but different names are *not* considered equal.

Because we translate the nominal recursive types of Haskell into structurally recursive types in the IL, we do not need to rely on the first property. Indeed, because our final aim is bytecode verification we should *not* rely on this property, since doing so could open an attack vector on the verifier. Nonetheless, a correctly constructed front-end which produces IL expressions will maintain the structural equality of Haskell types with the same name. However, without special care, the IL will fail to discriminate between two nominally different Haskell types which have the same structure, violating the second property.

We could choose simply to ignore this difference. After all, equating two structurally identical types can never violate memory-safety. On the other hand, a common programming idiom in Haskell is to define abstract data types by using the module system to hide the data type constructors. This way the programmer can encapsulate the data type behind an interface, a practice which has well-known software engineering benefits. If we ignore the nominal difference in the IL, it would be possible for an attacker to subvert such an abstract data type by creating a structurally identical type in another module and using it as a back door. Such back-door access to data types could also be used to defeat the guarantees provided by techniques such as lightweight static capabilities.

Therefore, we must somehow preserve the nominal distinction of Haskell types. The simplist way to do so is to modify the μ binder slightly so that it carries a label containing the original type name, fully qualified with the module name. When comparing two types for equality, the labels must match as well as the structure of the types. Together with sanity checks on modules to prevent name forging and to ensure that the internals of abstract types are only referenced in their defining module this simple measure should prevent abstract type subversion scenarios as outlined above.

For the sake of brevity, these extra labels are elided on terms appearing in this paper. However, the name of the bound variable is consistently chosen to match the type name being defined. As a first approximation, one could pretend that μ binders do not respect α-equivalence.

2.3.5 Primitives

Rather than spelling out and treating primitive types in the IL, we have decided to parameterize the IL over an arbitrary "primitive system." A primitive system is composed of the following items:

- a finite set of symbols called "primitive types"
- a finite set of symbols called "primitive functions"
- for each primitive type, a countable (or finite) set of symbols called "primitives" (The sets of primitives for each type are not required to be disjoint)
- a decidable relation $\vdash_{prim} p :: pt$ which holds iff the primitive p is in the set of primitives of type pt

$$\frac{(x, t) \in \Gamma}{\Gamma \vdash_{ty} x :: t} \qquad (\textsc{Ty-Var})$$

$$\frac{(\%x, t) \in \Gamma}{\Gamma \vdash_{ty} \%x :: t} \qquad (\textsc{Ty-LetVar})$$

$$\frac{\$x :: t \equiv m}{\Gamma \vdash_{ty} \$x :: t} \qquad (\textsc{Ty-Def})$$

$$\frac{\Gamma \vdash_{ty} m :: t_2 \qquad t_1 \,\triangleright\, t_2}{\Gamma \vdash_{ty} m :: t_1} \qquad (\textsc{Ty-Eval})$$

$$\frac{\begin{array}{c}(x, t_1), \Gamma \vdash_{ty} m :: t_2 \\ \Gamma \vdash_{k} t_1 :: k \qquad \textsc{KGood}(\ k\ )\end{array}}{\Gamma \vdash_{ty} (\lambda x :: t_1.\ m) :: t_1 \to t_2} \qquad (\textsc{Ty-Lam})$$

$$\frac{\Gamma \vdash_{ty} m_1 :: t_1 \to t_2 \qquad \Gamma \vdash_{ty} m_2 :: t_1}{\Gamma \vdash_{ty} m_1\ \ m_2 :: t_2} \qquad (\textsc{Ty-App})$$

$$\Gamma \vdash_{ty} (\texttt{error}\ [t]\ \text{“message”}) :: t \qquad (\textsc{Ty-Error})$$

$$\frac{(\mathrm{x}, k), \Gamma \vdash_{ty} m :: t}{\Gamma \vdash_{ty} (\Lambda \mathrm{x} :: k.\ m) :: (\forall \mathrm{x} :: k.\ t)} \qquad (\textsc{Ty-TyLam})$$

$$\frac{\Gamma \vdash_{ty} m :: (\forall \mathrm{x} :: k.\ t) \qquad \Gamma \vdash_{k} t' :: k}{\Gamma \vdash_{ty} m\ [t'] :: [\mathrm{x} \mapsto t']t} \qquad (\textsc{Ty-TyApp})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{ty} m_1 :: t_1 \qquad \Gamma \vdash_{k} t_1 :: \star \\ \Gamma \vdash_{ty} m_2 :: t_2 \qquad \Gamma \vdash_{k} t_2 :: k \qquad \textsc{KGood}(\ k\ )\end{array}}{\Gamma \vdash_{ty} \texttt{seq}\, m_1\ m_2 :: t_2} \qquad (\textsc{Ty-Seq})$$

$$\frac{\vdash_{prim} z :: p}{\Gamma \vdash_{ty} @[\,p\,]\{\,z\,\} :: @\{\!|\,p\,|\!\}} \qquad (\textsc{Ty-Prim})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{ty} m :: v \\ \Gamma \vdash_{k} t :: k \qquad \textsc{KGood}(\ k\ ) \\ t \,\triangleright\, u \qquad u \looparrowright v\end{array}}{\Gamma \vdash_{ty} \texttt{roll}[t]\ m :: t} \qquad (\textsc{Ty-Roll})$$

$$\frac{\Gamma \vdash_{ty} m :: u \qquad u \looparrowright v}{\Gamma \vdash_{ty} \texttt{unroll}\ m :: v} \qquad (\textsc{Ty-Unroll})$$

Table 6: Typing Rules for the IL, Part1

$$\frac{\Gamma \vdash_{ty} m :: \langle\!|\, t_0, \cdots, t_{n-1} \,|\!\rangle \qquad 0 \le i < n}{\Gamma \vdash_{ty} \pi_i \;\ m :: t_i} \qquad (\textsc{Ty-Proj})$$

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

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

$$\frac{\begin{array}{c}\vdash_{prim} f :: (p_0, \cdots, p_{n-1}) \Rrightarrow (q_0, \cdots, q_{r-1}) \\ \Gamma \vdash_{ty} m_i :: @\{\!|\, p_i \,|\!\} \qquad \text{for all } 0 \le i < n\end{array}}{\Gamma \vdash_{ty} @\{\, f \,|\, m_0, \cdots, m_{n-1} \,\} :: \langle\!|\, @\{\!|\, q_0 \,|\!\}, \cdots, @\{\!|\, q_{r-1} \,|\!\} \,|\!\rangle} \qquad (\textsc{Ty-PrimFunc})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{ty} l :: s \qquad \Gamma \vdash_{k} s :: \star \\ (x, s), \Gamma \vdash_{ty} m :: t \qquad \Gamma \vdash_{k} t :: k \qquad \text{KGood}(\ k\ )\end{array}}{\Gamma \vdash_{ty} (\texttt{let}\,\{\, x \;=\; l\,;\,\}\ \texttt{in}\, m) :: t} \qquad (\textsc{Ty-Let})$$

$$\frac{\begin{array}{c}\Gamma' = (\%x_0, t_0), \cdots, (\%x_{n-1}, t_{n-1}), \Gamma \\ \Gamma' \vdash_{ty} m :: t \qquad \Gamma' \vdash_{k} t :: k \qquad \text{KGood}(\ k\ ) \\ \Gamma' \vdash_{ty} m_i :: t_i \qquad \Gamma' \vdash_{k} t_i :: \star \qquad \text{for all } 0 \le i < n\end{array}}{\Gamma \vdash_{ty} (\texttt{reclet}\,\{\, \%x_0 :: t_0 = m_0; \cdots; \%x_{n-1} :: t_{n-1} = m_{n-1}; \}\ \texttt{in}\, m) :: t} \qquad (\textsc{Ty-RecLet})$$

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

$$\frac{\begin{array}{c}j_i \in \textsc{rng}(\gamma_p) \qquad \text{for all } 0 \le i < n \qquad \text{the } j_i \text{ are ordered} \\ \Gamma \vdash_{ty} m :: @\{\!|\, p \,|\!\} \qquad \Gamma \vdash_{ty} d :: t \qquad \Gamma \vdash_{ty} a_i :: t\end{array}}{\Gamma \vdash_{ty} \texttt{primcase}[\, p\,]\ m\ \texttt{default}\ d\ \texttt{of}\ \{\, j_0 : a_0; \cdots; j_{n-1} : a_{n-1} \,\} :: t} \qquad (\textsc{Ty-PrimCase})$$

Table 7: Typing Rules for the IL, Part 2

- a mapping from each function symbol to a primitive function type, written $\vdash_{prim} f :: (p0, \cdots, p_{n-1}) \Rrightarrow (q0, \cdots, q_{r-1})$
- for each function symbol f, an interpretation function δ_f
- for each primitive type pt, an invertible function γ_{pt}, which maps the primitives of pt onto a (not necessarily proper) subset of Z
- a proof that whenever $\vdash_{prim} f :: (p0, \cdots, p_{n-1}) \Rrightarrow (q0, \cdots, q_{r-1})$ and $\vdash_{prim} x_i :: p_i$ for all $0 \leq i < n$, then $\delta_f(x_0, \cdots, x_{n-1}) = (z_0, \cdots, z_{r-1})$ and $\vdash_{prim} z_i :: q_i$ for all $0 \leq i < r$

For example, let us define a simple primitive system with a few operations on the 32-bit two's complement integers. It shall have a single type symbol; let us call it "`Int`." This system defines the addition and subtraction functions on Int; let us assign these functions the symbols $+$ and $-$, respectively. The primitive symbols are then just the 32-bit binary sequences. We then set $\vdash_{prim} x :: \texttt{Int}$ whenever x is a 32-bit binary sequence. We further set $\vdash_{prim} + :: (\texttt{Int}, \texttt{Int}) \Rrightarrow (\texttt{Int})$. Similar for $-$. The interpretation function δ_+, unsurprisingly, implements two's complement addition, and δ_- performs two's complement subtraction. Finally, we set $\gamma_{\texttt{Int}}$ to be the usual semantic mapping for two's complement integers. The proof is omitted, but obvious.

Although the definitions given here are abstract, the basic idea behind a primitive system is actually quite concrete. The primitive system is intended to be used to allow basic numeric computations be mapped to actual hardware CPU instructions or to highly optimized special-purpose software libraries. The proof obligation just says that the types assigned to the function symbols are correct.

The evaluation and typing rules for the IL (found throughout this section) contain references to the items in the list above. We make the convention that the rules are written with a fixed, albeit arbitrary, primitive system in mind. In other words, the judgment rules should be read with the primitive system held as an abstract parameter.

The most interesting syntactic forms dealing with primitives are the primitive function form and the primitive case form. A primitive function application consists of a function symbol and some number of arguments. When the value of a primitive function is demanded, the arguments are evaluated in left-to-right order (although, in truth, the order is arbitrary). When all arguments have been reduced to primitives, the interpretation function for that primitive function symbol is called, and the primitive function redex is replaced by a product containing the results. Primitive function applications are always required to be saturated, but currying can be recovered by introducing lambdas.

The primitive case form is much like the regular `case` form, but it operates on primitives rather than sum types. Case arms are numbered, as for the `case`, but now the indices are drawn from Z rather than the non-negative integers. They are still required to be ordered. The case scrutinee of a `primcase` must be a primitive type. Once the scrutinee is evaluated to a primitive, it is interpreted via the primitive interpretation function γ_p, where p is the type of the primitive. Thereafter, evaluation is much the same as for `case`. If a case arm matches the interpreted primitive, that arm is chosen; if no arm matches then the default is chosen. Unlike the regular `case`, however, no arguments are passed into the `primcase` arm.

One final point to note is that the primitive type p is attached directly to the `primcase` syntactic form. This primitive type is *not* subject to substitution which makes `primcase` monomorphic with respect to the type of its scrutinee. We therefore maintain phase distinction because the primitive type of a `primcase` is fixed and appropriate code can be generated at compile-time.

2.3.6 `seq` and its consequences

Haskell can largely be thought of as a heavily sugared variant of the call-by-name lambda calculus. Indeed, all of the computational features we have examined thus far are straightforward extensions to the base calculus. However, Haskell also includes the seq primitive, which does not correspond to any call-by-name lambda term. seq is specified by the following equations: [17, 32]

$$\texttt{seq}\, x\, y = \begin{cases} \bot & \text{when } x = \bot \\ y & \text{when } x \neq \bot \end{cases}$$

seq is included in the language to solve a performance problem with Haskell. Because Haskell is a non-strict language, evaluation of functions is usually delayed until the runtime can determine that the value is actually needed for the computation. This is a big performance win when expensive, unneeded calculations are skipped entirely. It is also a win in terms of the expressiveness of the language; interesting algorithms can be expressed very naturally as the navigation of an "infinite" data structure. However, non-strictness also has its costs. It sometimes takes significantly more time and space to keep track of a long string of cheap, delayed computations than it would simply to perform them. Haskell programmers call such situations "space leaks," and they are probably the most common performance problem with Haskell programs.

To help deal with the problem of space leaks, the seq primitive function was introduced. Operationally, seq evaluates its first argument to a value before returning its second argument.[14] Using seq in the right places can cause the compiler to evaluate expressions earlier and allow the garbage collector to reclaim intermediate results sooner. This effect can significantly reduce the space usage of a program. It can also enable beneficial compiler optimizations which would otherwise be unsound.

However, seq is not expressible within the call-by-name lambda calculus and so it represents a true increase in expressive power over the base language. This has a number of unfortunate consequences, including the loss of sound η-conversion, the weakening of the parametricity theorem for the language, and the weakening of Wadler's "free theorems" [17].

Nonetheless, seq is widely regarded as indispensable. It is therefore included as a base syntactic form in the IL. The evaluation rules E-Seq1 and E-Seq2 give an operational interpretation. One subtle point to note is that the typing rule Ty-Seq restricts the first argument to have kind $\star$, which means that seq only operates on lifted data items. Products, because they are unlifted, are excluded.[15]

As with products and primitive applications, the IL seq form is syntax and must therefore be fully saturated; again, a curried version is recovered by introducing lambdas. The translation of the seq function from the Haskell Prelude can be given as:

$$\begin{array}{ll} \textit{\$seq} & ::\ \forall \text{A} :: \star.\ \forall \text{B} :: \star.\ \text{A} \to \text{B} \to \text{B} \\ & \equiv\ \Lambda \text{A} :: \star.\ \Lambda \text{B} :: \star.\ \lambda x :: \text{A}.\ \lambda y :: \text{B}.\ \texttt{seq}\, x\, y; \end{array}$$

2.4 Type Erasure

Haskell is a language with strong static typing which maintains *phase distinction* [14]. This means that type information is not needed at runtime and that no value reduction is required for typechecking. A natural way to compile Haskell is to perform *type erasure*, which removes all type information once it is no

[14] Actually, the Haskell Report does not specify that seq have any operational behavior and the description here is an over-specification. Seq is often implemented in exactly this way, however.

[15] It may be possible to further restrict seq in order to limit its negative semantic effects. The primary problem with seq is that it allows one to distinguish between $\bot$ and $\lambda x :: t.\ \bot$. If we can structure the kind system so that seq is unapplicable at function types, we conjecture that we would recover sound η-conversion and parametricity. To do this, we would require a kind system which distinguishes "ground" types, which have pointed domains, from function types, which would not. To retain the ability to have polymorphic functions, we would require subsumption in the kind system so that the ground kind and the function kind are subsumed by $\star$. It is not immediately clear what effect this semantic change would have when pushed back into Haskell, but it would probably mean that seq would become *syntax* in the language which would have to be applied to at least its first argument, rather than being exposed as a special polymorphic function, as it is in Haskell98.

$r ::=$	Type-Erased Terms
x	Term Variable
$\%x$	Let-Bound Variable
$\$x$	Top-Level Definition
$\lambda x.\ r$	Abstraction
$r\ \ r$	Application
`error` “message”	Error
`seq` $r\ r$	Seq
$@\{\, p \,\}$	Primitive
$@\{\, f \mid r_0, \cdots, r_{n-1} \,\}$	Primitive Function
$\langle\, r_0, \cdots, r_{n-1} \,\rangle$	Product
$\pi_i\ \ r$	Product projection
$\&C_i\ r$	Sum Constructor
`let` $\{\, x\ =\ r\,;\}$ `in` r	Non-Recursive Let
`reclet` $\{\, \%x_0 = r_0\,;$ $\cdots\,;$ $\%x_{n-1} = r_{n-1}\,;$ $\}$ `in` m	Recursive Let
`case` r `default` r `of` $\{\quad i_0 : arm_0\,;$ $\cdots\,;$ $i_{n-1} : arm_{n-1}\,;$ $\}$	Case
`primcase`$[\,p\,]\ r$ `default` r `of` $\{\quad i_0 : arm_0\,;$ $\cdots\,;$ $i_{n-1} : arm_{n-1}\,;$ $\}$	Primitive Case

Table 8: Syntax of type-erased terms

longer needed (sometime after typechecking). Most current Haskell compilers do this at some point; the Yhc compiler, for example, performs type erasure early and uses an untyped intermediate language for most of the compilation pipeline. Type erasure is also a natural way to give the dynamic semantics of the language. For our present aim, we wish to retain type information all the way up to code generation so that it is available for verification purposes. Nonetheless, a type erasure transformation for the IL holds some interest for reasons that shall made clear in what follows.

The syntax of type-erased terms is given in Table 8. It closely follows the term syntax of the IL except that type annotations are removed from binders and forms that deal solely with types are removed altogether.[16]

The evaluation rules for type-erased terms are omitted for brevity, but they can be straightforwardly generated from the rules for the IL by removing inapplicable rules and dropping type annotations from binders. Type erasure is a simple syntax-driven transformation and its definition is given in Table 9.[17]

To illustrate the effect of type-erasure, consider the following IL definition:

$$\$\textsc{oMu} \quad \doteq \quad \hat{\lambda}\mathrm{A} :: \star.\ \mu\,\mathrm{X} :: \star.\ \mathrm{X} \to \mathrm{A};$$

$$\begin{array}{lll} \mathit{\$omega} & :: & \forall \mathrm{A} :: \star.\ \mathrm{A} \\ & \equiv & \Lambda \mathrm{A} :: \star.\ (\lambda x :: \$\textsc{oMu}\ \ \mathrm{A}.\ (\texttt{unroll}\, x)\ \ x)\ \ (\texttt{roll}[\$\textsc{oMu}\ \ \mathrm{A}]\,(\lambda x :: \$\textsc{oMu}\ \ \mathrm{A}.\ (\texttt{unroll}\, x)\ \ x)); \end{array}$$

$omega defines the canonical non-terminating lambda term, decorated with the type annotations necessary to make it typecheck. The type-erased form of *$omega* is given by:

$$[\![\mathit{\$omega}]\!] = (\lambda x.\ x\ \ x)\ \ (\lambda x.\ x\ \ x)$$

[16] One minor exception is the `primcase` form, which retains some type information. Nonetheless, this maintains phase distinction and allows type erasure as discussed in section 2.3.5.

[17] It is our intention to define type erasure so that it commutes with evaluation. A proof to this effect would strengthen our arguments about type rewrite rules. Some of the unusual rules defining what constitutes a value in the IL derive from the desire for IL values to be mapped by type-erasure onto type-erased values.

With the obfuscating type annotations removed, this term should be immediately familiar.

Another way to think about type erasure is to turn the relation around and view it as function from type-erased terms to *sets* of IL terms. In this view, we can think of IL terms as type-annotated versions of the erased terms; let us call such terms the IL terms *generated* by the untyped term.[18] Suppose we are given a type-erased term x. Let us define the set S to be the set of IL terms generated by x (note that S might be empty). Let us further define the set $T_x = \{t \mid \emptyset \vdash_{ty} m :: t \text{ and } m \in S\}$. Then the set T_x represents all the types that could be assigned to x. We say that T_x is the *types-set* of x.

We can more directly define the types-set of a term x as:

$$T_x = \{t \mid \exists m. [\![m]\!] = x \text{ and } \emptyset \vdash_{ty} m :: t\}$$

Going a step further, we can define a notion of type subsumption based on these ideas. We say that a type s *subsumes* a type t iff for every type-erased term x such that $t \in T_x$ it holds that $s \in T_x$. The intention here is that every computation which can be assigned type t can also be assigned type s. Alternately, we may think of the (type-erased) terms of type s as a set, in which case the set of terms of type s includes the set of terms of type t (which may give a better intuition for the term "subsumption").

We write $s :> t$ when s subsumes t.

It should be clear from the definitions that subsumption is both reflexive and transitive, which makes it a preorder on types. It is not immediately clear whether subsumption in this system is anti-symmetric; establishing either a positive or a negative result for this question might be an interesting avenue for future work.

Finally, note that type erasure can be used as a proof device. See For example, Mitchell uses type erasure and PERs to prove type soundness and confluence properties for F_2 [25].

2.5 Type Rewriting Rules

The certificate-checking algorithm presented in section 5 relies on a number of type rewrite rules. These rules specify transformations that the verifier is allowed to make on the type of a data item. In this section we justify these rules using the ideas of type erasure and subsumption.

Let σ be a type rewrite rule. We write the application of σ to a type t as $\sigma(t)$. We say that a type rewrite rule is *valid* iff $\sigma(t) :> t$ for all t where the rule applies. In other words, a rewrite rule is valid iff it rewrites a type so that every term which can be assigned type t can also be assigned type $\sigma(t)$.

Note that rewrite rules are partial; they usually require that the type to be rewritten be in a particular form. The result of applying a rewrite rule to an inappropriate type is undefined. In the verifier, this is treated as an error.

In order to demonstrate a type rewrite rule is valid, it is sufficient to demonstrate a transformation on typed IL terms which preserves the type-erased image of the term and sends terms of type t to terms of type $\sigma(t)$. For each of the following rules, we shall give such a transformation and discuss why it has the necessary properties.

The definition of type rewrite rules and term rewrite rules are given as one or more inference rules. Many rewrite rules are parameterized, and these parameters will be used freely in the inference rules.

[18] We can easily classify the complexity generating the IL terms of an untyped term as as recursively enumerable. Sequentially generating all IL terms is trivial. Typechecking is decidable, so we can easily filter all IL terms to only those which typecheck. We further filter to only those IL terms which have the same type-erasure image as the target term. The resulting list is the desired list of generated terms. The problem is undecidable because we can trivially reduce the problem of Curry-style System F typechecking to it, which is known to be undecidable.

$$
\begin{aligned}
&\llbracket x \rrbracket &&= x\\
&\llbracket \%x \rrbracket &&= \%x\\
&\llbracket \$x \rrbracket &&= \$x\\
&\llbracket \lambda x :: t.\ m \rrbracket &&= \lambda x.\ \llbracket m \rrbracket\\
&\llbracket m_1\ \ m_2 \rrbracket &&= \llbracket m_1 \rrbracket\ \ \llbracket m_2 \rrbracket\\
&\llbracket \Lambda \mathrm{X} :: k.\ m \rrbracket &&= \llbracket m \rrbracket\\
&\llbracket m\ [t] \rrbracket &&= \llbracket m \rrbracket\\
&\llbracket \texttt{error}\ [t]\ \mathrm{msg} \rrbracket &&= \texttt{error}\ \mathrm{msg}\\
&\llbracket \texttt{seq}\ m_1\ m_2 \rrbracket &&= \texttt{seq}\ \llbracket m_1 \rrbracket\ \llbracket m_2 \rrbracket\\
&\llbracket \texttt{roll}[t]\ m \rrbracket &&= \llbracket m \rrbracket\\
&\llbracket \texttt{unroll}\ m \rrbracket &&= \llbracket m \rrbracket\\
&\llbracket \pi_i\ \ m \rrbracket &&= \pi_i\ \ \llbracket m \rrbracket\\
&\llbracket @[\,p\,]\{\,z\,\} \rrbracket &&= @\{\,z\,\}\\
&\llbracket @\{\,f \mid m_0, \cdots, m_{n-1}\,\} \rrbracket\\
&\quad = @\{\,f \mid \llbracket m_0 \rrbracket, \cdots, \llbracket m_{n-1} \rrbracket\,\}\\
&\llbracket \langle\, m_0, \cdots, m_{n-1}\,\rangle \rrbracket\\
&\quad = \langle\, \llbracket m_0 \rrbracket, \cdots, \llbracket m_{n-1} \rrbracket\,\rangle\\
&\llbracket \&C_i\ [t_0, \cdots, t_{n-1}]\ m \rrbracket\\
&\quad = \&C_i\ \llbracket m \rrbracket
\end{aligned}
$$

$$
\llbracket \texttt{let}\,\{\,x\ =\ l\,;\}\,\texttt{in}\,m \rrbracket \quad = \quad \texttt{let}\,\{\,x\ =\ \llbracket l \rrbracket\,;\}\,\texttt{in}\,\llbracket m \rrbracket
$$

$$
\begin{array}{l}
\llbracket \texttt{reclet}\,\{\,\%x_0 :: t_0 = m_0\,;\\
\qquad\quad \cdots\,;\\
\qquad\quad \%x_{n-1} :: t_{n-1}\\
\qquad\qquad = m_{n-1}\,;\\
\qquad \}\,\texttt{in}\,m\,\rrbracket
\end{array}
\quad = \quad
\begin{array}{l}
\texttt{reclet}\,\{\,\%x_0 = \llbracket m_0 \rrbracket\,;\\
\qquad \cdots\,;\\
\qquad \%x_{n-1} = \llbracket m_{n-1} \rrbracket\,;\\
\quad \}\,\texttt{in}\,\llbracket m \rrbracket
\end{array}
$$

$$
\begin{array}{l}
\llbracket \texttt{case}\ m\\
\quad \texttt{default}\ d\ \texttt{of}\\
\quad \{\quad i_0 : arm_0\,;\\
\qquad \cdots\,;\\
\quad\ i_{n-1} : arm_{n-1}\,;\\
\quad \}\rrbracket
\end{array}
\quad = \quad
\begin{array}{l}
\texttt{case}\ \llbracket m \rrbracket\\
\quad \texttt{default}\ \llbracket d \rrbracket\ \texttt{of}\\
\quad \{\quad i_0 : \llbracket arm_0 \rrbracket\,;\\
\qquad \cdots\,;\\
\quad\ i_{n-1} : \llbracket arm_{n-1} \rrbracket\,;\\
\quad \}
\end{array}
$$

$$
\begin{array}{l}
\llbracket \texttt{primcase}[\,p\,]\ m\\
\quad \texttt{default}\ d\ \texttt{of}\\
\quad \{\quad i_0 : arm_0\,;\\
\qquad \cdots\,;\\
\quad\ i_{n-1} : arm_{n-1}\,;\\
\quad \}\rrbracket
\end{array}
\quad = \quad
\begin{array}{l}
\texttt{primcase}[\,p\,]\ \llbracket m \rrbracket\\
\quad \texttt{default}\ \llbracket d \rrbracket\ \texttt{of}\\
\quad \{\quad i_0 : \llbracket arm_0 \rrbracket\,;\\
\qquad \cdots\,;\\
\quad\ i_{n-1} : \llbracket arm_{n-1} \rrbracket\,;\\
\quad \}
\end{array}
$$

Table 9: Type Erasure

2.5.1 Polymorphic Application

This type rewrite rule takes a polymorphic type and replaces it with a specialization. The rewrite rule itself is parameterized by the type to use for specialization. We write the polymorphic application rule at type s as PolyAp s.

We require that s be closed, well-kinded type. Let us call its kind k.

Then, the inference rule which defines PolyAp s is:

$$
\frac{\Gamma\ \vdash_{k} s :: k}{\forall \mathrm{X} :: k.\ t \mapsto ([\mathrm{X} \mapsto s]t)} \qquad \text{(PolyAp)}
$$

The associated term rewrite rule is:

$$
\frac{\Gamma\ \vdash_{ty} m :: (\forall \mathrm{X} :: k.\ m)}{m \mapsto m\ [s]} \qquad \text{(PolyAp-Tm)}
$$

In short, the term rewrite rule involves adding a type application. During type erasure type applications are removed, so this term rewrite rule is clearly erasure-image preserving. It is also not difficult to see that it has the correct type effect and that the resulting term is correctly typed.

2.5.2 Type-lambda Hoisting

Sometimes a polymorphic type may have a universal binder nested under the right side of one or more function arrows. We cannot use the PolyAp rule on such types, because the universal binder is not the outermost construct. However, it is possible to hoist these binders so that they do appear outermost. The rewrite rule that performs this rewrite is called PolyHoist.

$$\frac{t \mapsto t'}{(s \to t) \mapsto (s \to t')} \qquad (\textsc{PolyHoist-1})$$

$$\frac{\mathrm{x} \notin \mathrm{FV}(s)}{(s \to \forall \mathrm{x} :: k.\ t) \mapsto (\forall \mathrm{x} :: k.\ s \to t)} \qquad (\textsc{PolyHoist-2})$$

The term rewrite rules for PolyHoist are rather verbose, so we shall simply describe them here. The basic idea that we introduce a new type lambda at the outermost level of the term, binding a new type variable. We then traverse the original term, finding all maximal sub-terms which have a polymorphic binder outermost. We then apply these sub-terms to the new type variable.

Because the term rewrite only manipulates type-binding lambdas and type applications, it clearly preserves the type-erasure image of the term. Although we have not verified the proofs, we conjecture that term transformation will generate correctly-type terms, and that it will have the desired effect on the types of the terms.

2.5.3 Polymorphic Multi-application

The above two rules can be combined into a form of multi-application. The type rewrite rule PolyApply *xs*, where *xs* is a list of closed, well-kinded types, indicates a polymorphic rewrite of multiple types. If the universal binders are all on the outside of the term, then PolyApply simply corresponds to multiple applications of PolyAp. However, PolyApply will also travel down the right spine of function arrows to apply polymorphic types.

This rule can be seen as a composition of (multiple applications of) the above two rules, and thus it should be clear that it is a valid rewrite rule.

This rule is the one actually used by the verification algorithm to deal with polymorphic specialization because it is the most flexible.

2.5.4 Sum Expansion

In the IL, sum types are essentially finite maps from tag values to types. To correctly type, the only real restriction on these types is that the one corresponding to the *actual* tag of the data constructor match the type of the enclosed data. The types assigned to the other tags are largely irrelevant.

Sum expansion exploits this fact to rewrite sum types by adding new tag values to the sum. The types assigned to these tags can be any arbitrary valid type. We know this because the actual tag of the data constructor must be one of the tags already in the sum!

The ExpandSum i t rule is used to add the tag i, with type t to a sum type. The type t can be any type that results in a correctly typed sum (which essentially means any valid product type).

The ExpandSum rule is defined by:

$$\frac{i' \neq i_x \text{ forall } x \text{ where } 0 \leq x < m}{\{\!|\ i_0 : t_0, \cdots, i_{m-1} : t_{m-1}\ |\!\} \mapsto \{\!|\ i_0 : t_0, \cdots, i' : t, \cdots, i_{m-1} : t_{m-1}\ |\!\}} \qquad (\textsc{ExpandSum})$$

The associated term rewrite rule looks almost identical:

$$\frac{i' \neq i_x \text{ forall } x \text{ where } 0 \leq x < m}{\&C_j\ [i_0 : t_0, \cdots, i_{m-1} : t_{m-1}]\ z \mapsto \&C_j\ [i_0 : t_0, \cdots, i' : t, \cdots, i_{m-1} : t_{m-1}]\ z} \qquad (\textsc{ExpandSum-Tm})$$

Again, this term rewrite rule manipulates only portions of the term that are erased, so the rewrite preserves the type-erasure image. It should also be immediately clear that the rule generates a well-typed term and that it has the expected effect.

The actual rule used by the verifier is a slight generalization of this one which allows sums to be expanded by multiple components in one rewrite. It can be straightforwardly defined as multiple applications of the above rule.

2.5.5 Roll and Unroll

The final two rewrite rules used by the verification engine are closely related. These rewrite rules are used to witness the isomorphism between recursive types and their unrolling. These rules are used by the verifier to handle recursive types. During compilation, uses of the IL `roll` and `unroll` constructs are turned almost directly into applications of these rules.

First, we consider the RollType t rule. It is defined by:

$$\frac{t \looparrowright s}{s \mapsto t} \tag{\textsc{RollType}}$$

The corresponding term rewrite rule is:

$$\frac{\emptyset \vdash_{ty} m :: s \qquad t \looparrowright s}{m \mapsto \mathtt{roll}[t]\ m} \tag{\textsc{RollType-Tm}}$$

Next, the UnrollType rule is defined by:

$$\frac{s \looparrowright t}{s \mapsto t} \tag{\textsc{UnrollType}}$$

The corresponding term rewrite rule is:

$$\frac{\emptyset \vdash_{ty} m :: s \qquad s \looparrowright t}{m \mapsto \mathtt{unroll}\ m} \tag{\textsc{UnrollType-Tm}}$$

As before, we only manipulate constructs which are subject to type-erasure, and the rules must therefore preserve type-erasure images. That the rules generate well-typed terms with the necessary type rewrites is almost immediate from the definitions.

3 Compilation Target

In the previous section, we introduced the compiler intermediate language and presented a term-rewriting operational semantics. However, implementing Haskell programs via term-rewriting has major performance problems. Better techniques are required.

The most successful and widely used implementation technique for Haskell is based on the idea of the graph reduction machine, or G-machine. The basic idea behind the G-Machine is that a program is represented as a directed graph with a distinguished node representing the "root" of the graph. Each node of the graph is represents a part of the expression being evaluated. Execution proceeds by traversing the graph to find a reducible sub-expression and *overwriting* the root node of the entire sub-expression with the result of the reduction. This process proceeds until the root of the graph represents an irreducible value. See *The Implementation of Functional Programming Languages*, chapters 12–15 for an introduction to the topic [31].

The G-machine technique is successful because: the program graph can be naturally expressed in terms low-level concepts such as heap-allocated memory blocks and pointers, the graph concept lends itself well to garbage collection, G-machine combinators can be compiled into fixed strings of instructions, and graph reduction captures the work-sharing aspects of lazy evaluation.

Some Haskell compilers, such as GHC, use the G-Machine model as an intermediate step along the way to native machine code. Yhc, however, targets a virtual machine that directly implements G-machine reduction. The Yhc runtime is a combination of a G-machine and a stack-machine model. Combinators are defined using an imperative, stack-machine bytecode set somewhat like the Java bytecode set, but the overall execution of the program is directed by graph reduction.

The following sections describe in more detail the how the Yhc runtime executes bytecode programs. Much of the information in these section is taken from the Yhc documentation wiki: `http://www.haskell.org/haskellwiki/Yhc`.

3.1 Yhc Virtual Machine Architecture

At the highest level, the Yhc virtual machine is a von Neumann architecture with special support for graph reduction. To run a Yhc program, one designates a "main" module. The Yhc runtime finds this module on disk and loads it into memory. It then finds and loads all the (transitive) dependencies of the main module. After all program text is loaded, a block of memory of a predetermined size is allocated. This block will hold both the virtual machine stack and heap.[19]

The VM has a number of "global registers," which contain important program state. The most important of these is the instruction pointer, which keeps track of which bytecode instruction should be executed next. Registers also exist to keep track of the top of the stack, the location of the current stack frame, and a variety of other bookkeeping information.

The program stack contains a series of stack frames. Stack frames are pushed when combinators are evaluated and popped when a combinator finishes executing, in much the same way that stack frames are managed in a conventional language such as C. Each stack frame contains information necessary to restore the previous frame, graph node currently being evaluated, and the "working" stack which is manipulated by bytecode instructions.

The heap contains the program graph and is subject to garbage collection. Just before execution begins, the runtime builds a heap node corresponding to the main entry point of the program and sets up a stack frame to evaluate that node. As the program executes, the graph will evolve in the heap. The VM instruction

[19] Note that the VM stack and heap differ from the stack and heap of the C runtime itself, which are governed by the C ABI. Unless otherwise stated, the terms "stack" and "heap" shall refer to the VM stack and heap.

pointer is set to the beginning of the bytecode sequence for the entry point. Execution finally begins when the runtime enters its main loop and starts to interpret the program bytecode.

3.2 Yhc Heap Layout

The main data structure used by the Yhc runtime is the heap node. Each heap node contains a header followed by some number (possibly 0) of arguments. The header and arguments are each one machine word in length. The node header gives the runtime the information necessary to interpret the node arguments and determine the size of the node.

The node header is a combination of an info pointer and two flag bits. The info pointer is obtained by masking out the lower two flag bits. The flag bits indicate the node state, which may be one of the following four states:

- 0 – IND This node is an indirection node. It has no arguments and the info pointer points to another heap node.
- 1 – NORMAL This node is a normal node, and the info pointer points to an info node.
- 2 – GC Used by the garbage collector (see the Yhc documentation for details).
- 3 – HOLE Indicates a node which is currently under evaluation. The info pointer points to an info node.

Indirection nodes are used whenever it is necessary to "overwrite" a heap node with a new value. Rather than copy the new value into the old heap node (because it might not fit), an indirection node is used to point to it. When garbage collection occurs, indirection nodes are removed.

The remaining heap nodes fall into two categories: "constructors" and "applications." A constructor is an evaluated data item, such as a primitive integer or a algebraic data constructor. A constructor node is created when the value is first calculated and remains in the heap until it becomes unreferenced and is garbage collected. Constructor nodes are immutable and always have state NORMAL. Application nodes are further broken down into "partial applications" and "saturated applications" (or "thunks"). Partial application nodes always have state NORMAL. Thunks begin in state NORMAL. If they are required for evaluation, they enter state HOLE. When the evaluation of a thunk is complete, it is overwritten with its value, and thus enters state IND.

For constructors and applications, the info pointer will point to an "info node" which has further information about the node. For constructors, it will indicate if the node is a primitive or an algebraic data constructor. If the node is an algebraic constructor, it will additionally contain the name, arity, and tag number of the constructor. Info nodes are shared among all similar constructors. For Application nodes, the info pointer will point to a data structure describing the function to which the node arguments should be applied. Function info nodes contain the function's name, arity and some other housekeeping information. It will also contain pointers to the function's bytecode instructions and constant table. For partial applications, the info node will also indicate how many arguments are still needed to saturate the application.

3.3 Combinator Execution

The heart of the Yhc interpreter is the main loop which evaluates executes the body of combinators. When the value of a thunk is demanded, a new stack frame is pushed in order to evaluate the application. The local working stack starts empty and the arguments to the application become available as the combinator arguments of the current stack frame. Execution proceeds by reading the bytecodes and performing the associated actions. Some bytecodes may demand the values of other data in the heap. If the value of a thunk is demanded, a new stack frame is pushed and that thunk is evaluated before the current stack frame can continue. When the runtime encounters an instruction that causes a return, the top value on the working stack is used to overwrite the value of the original thunk. The current stack frame is then popped and execution proceeds with the next stack frame on the program stack. If the last stack frame has been popped, execution halts.

The full bytecode set for the Yhc runtime is given in the appendix, along with a description of the behavior of the instruction when executed. A simplified and abstracted bytecode set is given in Table 10. This simplified bytecode set is the initial target for IL lowering.

PUSH i	Read the ith value from the stack and push it onto the top.
PUSH_ARG i	Read the ith 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 x	Create a data constructor node with the name nm, tag number i, and arity x using the top x elements on 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 x	Take the top element off of the stack, pop the next x elements, and then replace the top element.
POP x	Pop the top x elements off of the stack.
ALLOC x	Create x "place-holder" nodes and push references to them on the stack.
UPDATE x	Overwrite the node pointed to by the $x + 1$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.
INT_SWITCH l ls	ls is a list of (value,label) pairs. Examine the top element of the stack, which must be an integer. If it matches any value from ls, jump forward to the specified label. If no value matches, jump forward to l.
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 an reference to the named super-combinator onto the stack.
PUSH_PRIM p	Push the given primitive symbol onto the stack.
PRIM_TO_INT pt	Take a primitive value off the top of the stack and push it's integer equivalent onto the stack, using the γ_{pt} function for the given primitive type.
CALL_PRIMITIVE f	Call the interpretation function for the primitive function symbol f, using the appropriate number of arguments from the top of the stack. Push the return value(s) onto the stack.
MK_BOTTOM msg	Create a closure which will abort the program and print the given message when evaluated.
REWRITE_TYPE σ	Rewrite the type of the top element on the stack using the given type rewrite rule. See section 2.5 for information on rewrite rules.

Table 10: The Simplified Bytecode Set

4 Compiler Implementation

We have implemented a compiler which takes IL as source code and produces certified bytecode as output. The implementation process occurred in parallel with the design of the IL and certificate checking algorithms; this helped to uncover errors or omissions in the design. In this section, we shall cover the implementation strategy at a high level and provide additional details for any parts which are novel or which were especially tricky to implement. We shall also make special note of how type information travels through the later stages of compilation, as this process is different from many other compilers which discard type information before these stages occur.

Our primary aims when developing this compiler were, in order of priority:

- to develop a correct implementation,
- to develop a modular solution so that each step can be understood and inspected independently, and
- to reuse, to the extent possible, techniques from the literature.

An additional meta-goal of this entire project is to try to close the gap between formal type theory and the practice of compiler construction. Thus, I have attempted to formulate each piece of the compiler with an eye toward the possibility of formalizing machine-checked proofs of correctness.

Absent from the above list is the goal of producing efficient code. We have purposefully neglected this important aspect of compiler implementation in order to keep the initial development manageable. Also, the primary point of interest in this project is the process of code and certificate generation, which works in much the same way regardless of the presence or absence of aggressive optimization. For these reasons, any optimizations which are not on the direct compilation path have been avoided, especially those which require complicated analyses.

The IL compiler can be viewed as three basic parts: the front-end, the lambda-lifter, and the back-end. In what follows, we shall examine each section in turn.

The implementation of the compiler, certificate checker and a small shell environment for interacting with the various stages of compilation required slightly less than 10,000 lines of Haskell code, including comments and whitespace.[20] The source, when compressed using gzip, is about 62K. The compiler was written using minor extensions to the Haskell 98 standard and with heavy reliance on GHC's standard libraries.

4.1 The Front End

The front-end of the IL compiler has three major tasks: parsing, serialization, and type checking.

Because the IL is an intermediate language, the human-interface concerns which dominate the concrete syntax design for most languages do not carry as much weight as usual. Therefore, there is little syntax sugar and no type inference engine to ease the burden of the many type annotations. The concrete syntax of the IL is designed, as much as possible, to be a direct representation of the abstract syntax. Each construct of the IL has a distinctive sequence of characters in the concrete syntax. The concrete syntax is designed to be LL(1) and the parser is implemented using simple recursive-decent techniques from the Parsec parsing library [20].

Following the implementation techniques of Pierce [36], the internal representation of the IL uses de Bruijn indices rather than explicit nominal binders. By using a nameless de Bruijn representation, we have avoided the problems arising from α-conversion. This makes the implementation of capture-avoiding substitution much easier. Normally, a compiler would not be very concerned about substitution. However, we wished to use the same program representation to directly interpret IL terms as well as for compilation. Direct

[20] However, and external projects were used to read and write bytecode and to drive the shell environment.

interpretation of the IL terms helped us to define the operational semantics of the IL, and gave an additional way to verify the results of compilation. Using a recursive decent parser makes the process of determining variable scope and translating named variables to indices very easy. In order to support serialization, IL constructs which bind variables save the original name of the bound variable.

Serialization is a straightforward affair which involves a simple tree walk of the abstract syntax. The only interesting part of this process involves disambiguating variables which share a scope and have the same name. The serializer maintains an environment containing the names of all variables currently in scope, and it simply adds numeric indices to variables which have the same name as a variable already in scope.

Two different serializers exist: one which produces the concrete syntax of the IL and can be viewed as an inverse to parsing, and one which produces a representation of the abstract syntax using LaTeX macros. This second serializer is used to produce much of the IL code appearing in this paper.

The final task of the front-end is typechecking. The base algorithm again follows Pierce [36]. The Church-style calculus enables a simple top-down algorithm which does not require unification. The type-checking rules given in tables 5, 6, and 7 are nearly algorithmic; this is no accident, because they were transcribed from the code of the typechecker implementation.

The only point of particular note about typechecking has to do with checking an entire IL module. One must take care that type definitions do not refer to themselves (directly or indirectly) because this would allow one to define unrestricted equi-recursive types and destroy the strong normalization property of the type system. To ensure that this does not occur, type definitions are sorted topologically; if any definition completes a cycle, the module is rejected.

4.2 Lambda Lifting

The G-Machine is a framework for performing combinator reduction. Although it is possible to compile the lambda calculus into a fixed set (or basis) of primitive combinators, such techniques tend to generate a large number of combinators. Instead, modern G-Machine implementations of Haskell compile lambda expressions into a custom set of combinators. Generating custom sets of combinators tends to produce programs using many fewer combinators. As program execution time is strongly influenced by the number of combinator reductions, using fewer combinators is a big advantage.

The combinators generated for the G-Machine are of a special form and are known as "super-combinators" [15]. The defining characteristics of super-combinators are that they are combinators and that they contain only sub-expressions which are also combinators.

The process of translating unrestricted lambda expressions into super-combinators is known as lambda-lifting, and it is a crucial process for most modern Haskell implementations. Lambda lifting involves transforming an expression into some number of combinator definitions such that no definition contains any free variables.

An additional transformation that is closely related to lambda-lifting is the full-laziness transform. We say that a reduction technique has full-laziness if every expression is evaluated at most once. This property is generally regarded as desirable and is a feature of some Haskell compilers, either in full or in a limited form.[21] The full-laziness transform has been associated with lambda-lifting since Huges presented the two techniques together in a single, unified transformation [16].

However, Peyton-Jones and Lester showed that the full-laziness and lambda-lifting transformations could be treated separately [33]. The key idea involves replacing maximal free expressions by trivial let expressions before using a simple lambda lifter. For example, if e is a maximal free expression, then e would be replaced by $\mathtt{let}\,\{\,v \;=\; e\,;\}\,\mathtt{in}\,v$ where v is a fresh variable. The let definition is then "floated" outward until it is

[21] It is sometime desirable to have slightly-less-than-full laziness. If an expression is "small enough," it may be worthwhile execute the expression multiple times but avoid memory accesses.

just inside the innermost lambda that binds one of its free variables. Following let-floating, a simple, naïve lambda lifter is applied. This combination of transformations ensures the full-laziness property when the combinators are reduced in a G-Machine.

The IL compiler implements a lambda-lifter by following closely the implementation techniques of Peyton-Jones and Lester [33]. However, the techniques had to be adapted to our typed setting because the calculus in the paper was typeless. In most ways the adaptation was straightforward; however, using a typed calculus did present one major difficulty. During the let-floating phase, it is possible to float a let definition outside a type lambda which binds one of its free type variables. An expression so transformed will not typecheck.

There are two basic ways to resolve this difficulty. The first is to simply stop floating a let definition as soon as it hits either a term *or* a type lambda which binds one of its type or term variables. This solution is unsatisfactory because it compromises the full-laziness property to no advantage and can lead to asymptotic slowdown of the compiled programs. The second solution, which is employed in the IL compiler, is to abstract over free type variables when transforming maximal free expressions into let bindings. For example, if an expression e contains the type variable A with kind k, then e would be replaced by $\texttt{let}\,\{\, v \,=\, \Lambda\mathrm{B} :: k.\ [\mathrm{A} \mapsto \mathrm{B}]e\,;\}\,\texttt{in}\, v\ [\mathrm{A}]$ where v and B are fresh variables. This allows the let definition to float over the type lambda which binds A and still typecheck properly.[22]

Unfortunately, we cannot simply abstract over all free type variables in expressions we wish to float. Instead we must abstract *only* the type variables bound by type lambdas over which the expression will actually have to float. Otherwise, typechecking will fail in some cases.

The problem is that term-lambda-bound variables may have types which contain free type variables. However, there is no way to abstract the free type variables in the type of a term variable once it is bound. Thankfully, if such a term variable appears in a maximal free expression, it is impossible to float over the type lambda which binds the free type variable (because the type lambda must appear outside the term lambda, and we will stop floating once we reach the term lambda).

In order to make sure we only abstract over the required type variables, extra information about which type variables are bound inside which term lambdas needs to be maintained during maximal free expression identification. This complicates the analysis somewhat.

Happily, the identification of maximal free expressions and the handling of free type variables is the only significant complication. Implementing the the naïve lambda lifter was completely straightforward. We found that the modular approach advocated by Peyton-Jones and Lester helped tame the complexity of the typed lambda lifter. If one were required to incorporate all of the minor changes required as well as the large change discussed above into a single monolithic transformation, it would quickly become difficult to manage.

4.3 Code Generation Stage 1

With lambda-lifting completed, the original IL code has been transformed into super-combinators. In this form type and term lambdas appear only as the outermost constructs of a definition. At this point, we strip off the lambdas and instead think of variables as being bound "by the combinator." We call the number of term lambdas that are absorbed into the combinator definition in this way the combinator's "arity," and it represents the number of parameters that must be supplied to the combinator before it can be reduced.

Now that free variables and lambdas have been removed from super-combinator definitions, it becomes possible to assign a specific sequence of stage 1 G-Machine bytecode to each syntactic construct of the IL. The stage 1 bytecode set is fairly small, and has been reproduced in full in table 10. Like the IL, stage 1 bytecode operates on explicitly-typed data. Unlike the IL, however, the type of a data item is not unique, in much the same way a type-erased IL term can sometimes be assigned multiple types. Thus, there is an

[22] Interestingly, this transformation also tends to introduce large amounts of higher-rank polymorphism. This transformation would be impossible if one instead used an IL which lacked rank-n polymorphism.

explicit instruction for manipulating the types of expressions on the program stack, called `REWRITE_TYPE`. The argument to this instruction is a type rewrite rule. These rules are each each related to a syntactic construct of the IL which manipulates types. See section 2.5 for more details about these rewrite rules.

All of the remaining IL constructs can be translated in a straightforward way into the computational instructions of the stage 1 bytecode. Below we highlight some of the interesting points of this transformation process.

The translation of term variables is perhaps the trickiest part of this process because it interacts strongly with the translation of term products and the `let` and `reclet` constructs. The basic idea is that each variable is either a lambda-bound variable, a `let`-bound variable, or a `reclet`-bound variable. Each lambda-bound variable is translated into a reference to one of the super-combinator parameters via the `PUSH_ARG` instruction. `let` and `reclet` expressions are translated by first building the right-hand-side expressions on the stack and then translating the body. References to the bound variables are translated into references to specific stack locations via the `PUSH` instruction. The argument or stack position of each variable is tracked by using an environment which maps each de Bruijn index to an argument or stack position.

However, keeping track of the environment is complicated by term products. We wish term products to have no explicit runtime representation, so products are organized as contiguous arrays of either combinator parameters or stack positions. For example, an expression such as $\lambda x :: \textsc{a}.\ \lambda y :: \llparenthesis\, \textsc{b}, \textsc{c}, \textsc{d}\, \rrparenthesis.\ \cdots$ will be translated into a combinator with four parameters where x is bound to the first parameter and y bound to the second, third and fourth parameters. Products built on the stack are similarly organized into contiguous sequences of stack slots. The nice thing about this organization is that product projection is translated very simply as an offset calculation from the base of the product.

Fortunately, we have anticipated this problem by designing the kind system to differentiate products from other sorts of data. We can use the kind of an expression to drive its bytecode translation. Our de Bruijn index environment must simply maintain the base location of each variable and its kind. From this information, the correct argument or stack location can be calculated without difficulty.

References to top-level bindings are directly translated into `PUSH_FUNC` instructions. Term application is handled by first building the right side of the application on the stack, then the left side, and finally emitting the `APPLY` instruction. This has the effect of building all the arguments to a string of applications on the stack with the rightmost arguments on the bottom and then emitting multiple `APPLY` instructions in a row. In later stages, these multiple `APPLY` instructions will be translated into more efficient instructions which can apply multiple arguments at once.

The translation for case expressions involves three basic steps. First, code is generated for the case scrutinee. The scrutinee is then `EVAL`ed to force the scrutinee expression to weak head-normal form. A `LOOKUP_SWITCH` expression is then emitted which will cause the G-Machine to branch based on the tag value of the scrutinee expression. Next, code is emitted for each case arm. This involves emitting a `LABEL` at the start of each arm and an `UNPACK` instruction to make available the product components of the sum expression. These values are then applied to the body of the case arm. Finally, the default branch of the case is emitted. This involves emitting a default `LABEL`, `POP`ing the case scrutinee value, and then emitting code for the default branch.

During this stage, the primitive system is uninterpreted and is translated into the place-holder instructions `PUSH_PRIM`, `PRIM_TO_INT` and `CALL_PRIMITIVE`.

The `seq` syntactic form is handled easily by the G-Machine environment. All that is required is to first generate code for the first argument to `seq` and then emit the `EVAL` instruction. After the `EVAL` returns, the result is popped off the stack and evaluation continues with the second argument.

The most complicated translation is that of `reclet` expressions. Local recursion is modeled in the G-Machine by creating program graphs which contain cycles. This requires some way to reference a part of the heap before its definition has been completed. The way this is handled is by using the `ALLOC` and `UPDATE` instructions. `ALLOC` reserves space in the program heap, but leaves it uninitialized, and pushes a reference to this space on the program stack. The UPDATE instruction is used to overwrite the value of an uninitialized heap location. To translate reclet we first ALLOC a location for each right-hand-side. We then generate code for each RHS in turn. Once a RHS has been generated, we use UPDATE to overwrite the heap node we reserved for the purpose. Once all the reclet definitions have been created, we can proceed to generating the body of the reclet.

The final point of note about stage 1 code generation is that the control flow graph of the generated bytecode is completely tree-structured. There are two factors that contribute to this. First, the two control-flow instructions LOOKUP_SWITCH and INT_SWITCH only allow forward jumps. This forces the control-flow graph to be acyclic. However, we additionally enforce tree-structure by performing a lightweight continuation-passing-style transform as byte code is being generated. The CPS transformation causes any code generation patterns that would otherwise generate joins in the CFG to instead cause duplicated code patterns to be generated underneath case arms. While this might not be strictly necessary, it simplifies later analyses greatly. Furthermore, the potential for code duplication is rather small because the lambda-lifting phase tends to cause large functions to be broken down into much smaller pieces.

4.4 Code Generation Stage 2

There are three major changes that occur during the translation from stage 1 to stage 2. Stage 1 code generation targets a G-Machine which explicitly manipulates types during execution, similar to the way System F has explicit reduction rules relating to types. During the translation to Stage 2 code, we remove all REWRITE_TYPE instructions from the code stream to target a typeless G-Machine. If we were doing type-erasure, we would simply forget this type information. However, we are instead interested in doing "type segregation," where the type information is retained, but separated from the computational information. This separated type information becomes the type certificate. Certificates are discussed in more detail in section 5.

The second major change is that stage 2 bytecode has a baked-in primitive system corresponding to the capabilities of the Yhc runtime, whereas stage 1 bytecode is still conceptually parameterized over an arbitrary primitive system. Thus the stage 2 translation must map the generic primitive instructions of stage 1 onto stage 2 instructions corresponding to specific primitive operations. This means that the transformation from stage 1 to stage 2 bytecode is itself parameterized by a set of functions that translate the PUSH_PRIM, PRIM_TO_INT and CALL_PRIMITIVE instructions into sequences of stage 2 instructions. These additional transformation functions are what implement the primitive system in the compiler.

The third major change involves compressing sequences of stage 1 APPLY instructions into single stage 2 instructions. The primary reason for this change is efficiency of the generated code. APPLY is an expensive operation that involves a check for sufficient heap space,[23] the allocation of space for a new closure, and the copying of all the arguments of the previous closure to the new one. It is therefore desirable for the final code to contain as few APPLY instructions as possible.

The first and most simple way to compress APPLY instructions involves adding a numeric argument. This argument indicates how many stack items to apply. The stage 2 instruction APPLY x is equivalent to x stage 1 APPLY instructions in a row. However, at run-time, it will only perform a single free space check and only allocate a single new closure. The second way to compress APPLY instructions is to remove them altogether. Whenever a sequence of APPLY instructions immediately follows a PUSH_FUNC instruction, we can eliminate some or all of the following APPLYs. If there are at least as many APPLYs as the arity of the function, we can simply create a new, saturated application to the named function all at once using the new stage 2 MK_AP instruction. This instruction only allocates the closure once, and does not require a free space

[23] The APPLY instruction is special, in that is the only instruction apart from NEED_HEAP that causes a free space check and may trigger garbage collection. The reason for this is that it is not possible to staticly determine the amount of space required by an APPLY instruction. The amount of space required is a function of the arity of the underlying super-combinator, but this information is not captured by the type system and so requires a run-time check.

check. If there are fewer `APPLY`s than the arity of the function, then we use the `MK_PAP` instruction, which create a partial application to the named super-combinator using as many arguments as are available. Like `MK_AP`, `MK_PAP` does not require a free space check. Unfortunately, this compression of `APPLY` instructions complicates type certificate construction. Section 5.2 discusses these complications.

4.5 Code Generation Stage 3

The third stage of code generation takes care of the final few details required to target the actual Yhc bytecode instruction set. This primarily involves generating "constant tables" and converting label references to jump offsets. The Yhc bytecode format associates with each super-combinator definition a constant table, which is just a list of constants referenced in that combinator. Function names, constructor names, numeric constants and string constants are all referenced via the constant table. Where stage 2 code has instructions like `PUSH_INT` and `PUSH_STRING`, stage 3 code simply has the single instruction `PUSH_CONST`, which references the constant table. Also, stage 2 instructions which take names, like `MK_AP` and `MK_CON`, reference the constant table instead in stage 3.

Both of these transformations are simple and have no particular theoretical interest; they relate merely to details of the particular file format decision made by Yhc. For this reason, we shall not discuss these matters in detail.

5 Certificate Creation and Checking

The full type certificate required for verifying a bytecode module consists of the following items:

- the definitions of all type synonyms used in the module,
- the type and arity of each super-combinator in the module, and
- for each super-combinator in the module, a list of rewrite rules to apply after each bytecode instruction

The type synonym definitions are retrieved directly from the IL module and need no further processing. Type information about each super-combinator is available as soon as the lambda-lifting phase is complete. The lists of rewrite rules are extracted during the translation from stage 1 to stage 2 bytecode. This process will be explained in more detail below.

At the highest level of detail, verification follows these steps:

- Ensure that all identifiers refer to type synonyms, functions or data constructors (as appropriate) that are in scope.
- Ensure that type synonym definitions are acyclic.
- Ensure that each type synonym defines a closed type and that it kindchecks.
- Ensure that the module does not reference unsafe primitives or foreign code via the Foreign Function Interface.
- Ensure that each super-combinator is assigned a closed type which kindchecks and that is has at least as many function arrows as its arity.
- Check that the bytecode for each super-combinator actually implements a function of the given type.

Except for the final step all these steps are essentially sanity checks to make sure the module and its certificate are properly formed and do not require access to the bytecode stream.

Note that we disallow access to to runtime primitives and to foreign functions. This is because there is no good way to ensure that primitives and foreign code are used safely; we simply cannot allow untrusted code access to unsafe mechanisms that might be misused. We must relegate access to such facilities to trusted modules and only make them available to untrusted code via safe APIs if we are to achieve the desired security benefits.

5.1 Verification by Abstract Interpretation

The final step of verification is the most complicated and also the most interesting. 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 all paths through the control flow graph end by returning a result of the expected type. If all bytecode sequences in the module have their expected types, then verification succeeds and we conclude that super-combinator definitions have the types stated in the certificate.

We can view this procedure as automatically verifying a proof. The proof we wish to verify is the statement that all super-combinators perform computation that corresponds to the types given in the certificate. This proof proceeds by induction over the ascending Kleene chain that forms the semantics of the recursive function definitions. The base case of the proof is always trivial because the least element of the semantic domain corresponds to the non-terminating computation, which can be assigned any valid type. The verification procedure corresponds to the inductive step of the proof with the super-combinator type definitions forming the induction hypothesis.

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 `PUSH_CONST` 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 `ADD_W` instruction adds two primitive integers; for it to execute properly, there must be two fully-evaluated primitive integers on the top of the stack.

To perform verification, we first allocate an array to hold the results of the abstract 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. For most instructions, this 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, then the array is filled with an error symbol rather than a valid next state and verification will fail. The preconditions and stack effects for most instructions are readily apparent from a description of their function. A brief description of the stack effect of each instruction is given in the appendix. However, the more subtle cases will be discussed in detail below.

5.2 Incorporating Type Rewrite Rules

Now that we have laid out the basic ideas of the verification algorithm, the function of the type rewrite rules can be explained. As we have mentioned, 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 more specific 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, we once again write an error symbol into the array and verification will fail.

However, there is one problem with this scheme. Because we compress multiple `APPLY` instructions into fewer during stage 2 translation, it can happen that type rewrite rules appearing between consecutive `APPLY`s get "squeezed" out. These rules cannot simply be appended to the resulting instruction because these inner rules are supposed to operate on the intermediate closures that we now avoid creating.

Because of the constraints of the type system, the only type rewrite rules that can appear between two consecutive `APPLY` instructions are the `PolyApply` rule and the `UnrollType` rule. If we encounter an `UnrollType` rule, we simply break the compression process and emit two `APPLY` instructions, with the unroll rule attached to the first. We anticipate that this will occur infrequently and that the performance implications will be low. However, we expect that the `PolyApply` rule will often be intermixed with the `APPLY` instruction. It is important to be able to handle this case without interrupting the APPLY compression. Therefore, in addition to the normal list of rewrite rules attached to each instruction, we also attach a list of types at which to instantiate polymorphic parameters. This list is required to be empty for all instructions except APPLY, MK_AP and MK_PAP. For these instructions, we use the attached types to specialize universal types appearing in the function *before* arguments are consumed from the stack. After arguments are consumed, type rewrite rules are applied as usual. This correctly emulates the behavior of the typeful stage 1 G-Machine.

5.3 The Role of Status Codes

We have thus far described how an abstract representation of the types on the stack at each program point are calculated. However, some instructions require some 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 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 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 INT_SWITCH and FROM_ENUM) or perform primitive operations (such as ADD_W) 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, and accessing that location generates an error. The compiler inserts zaps in certain situations where it can staticly determine that data will no 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, many 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 reclet, as discussed in section 4.3. However, if its use was unchecked 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.[24]

Finally, note that the status codes record staticly-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*

[24] 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.

which is known to point to evaluated data. There is no contradiction, for example, if one has two pointers 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.

5.4 Managing Reserved Heap Space

There are a large number of Yhc instructions that require heap allocation. It is important that the runtime be able to manage its heap efficiently. The default runtime for Yhc is written in C and 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 reset to 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` instructions "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.[25] 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. Because of the way Yhc's memory is laid out, this would involve overwriting parts of execution stack.[26] 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 to 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.

5.5 Control Flow with Type Rewrite Effects

Most instructions have very straightforward abstract effects. All straight-line (i.e., 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 `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 instructions, in terms of their abstract effects, are `TABLE_SWITCH` and `LOOKUP_SWITCH`. The `TABLE_SWITCH` instruction is really just a compressed special case of the `LOOKUP_SWITCH`, so we will concentrate on the more general `LOOKUP_SWTICH`. Both instructions work 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. These instructions are 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

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

[26] Note that because the execution stack is organized to grow toward the heap, we must reserve enough space for the current stack frame as well as all heap allocations.

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 TABLE_SWITCH and LOOKUP_SWITCH instructions is that they rewrite the type of the examined data item to reflect the knowledge gained by examining the data constructor's tag. For example, if one examines a data item with the type $\{\!|\, 0 : \mathrm{A}, 1 : \mathrm{B} \,|\!\}$ and discover that the constructor tag is 0, then the type type $\{\!|\, 0 : \mathrm{A} \,|\!\}$ can be safely assigned to the data item 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.[27]

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 TABLE_SWITCH or 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 exactly when the type being unpacked is a sum type with exactly one variant, and type of this form are therefore required for UNPACK.

[27] 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.

6 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 [21]. 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 systems since its inception in the early 1990's. The verification algorithm is specified using prose in *The Java Virtual Machine Specification* [23]. However, because the process is rather tricky, a number of researchers have attempted to formalize the Java verification problem and prove the correctness of verification algorithms. Initial work focused on the most difficult areas of the Java verification problem, the handling of subroutines and object initialization [39, 8]. Freund and Mitchel followed up by expanding the type systems developed for the subroutine and object initialization problems into a system capable of handling almost all aspects of the Java verification problem [7]. In his doctoral thesis, Klein presented a system with similar scope which he formally verified in a theorem prover [19]. Klein's system differs from previous work (and is similar to ours) in that he focuses on a verification system using certificates (which he calls "lightweight verification") rather than performing type inference at verification time.

Although early implementations of the Java verifier had some problems [5], 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 originates with Java.

We cannot directly reuse previous work on Java bytecode verification because the execution model of Haskell is so different from that of Java. While the Yhc bytecode set is stack based (like Java), it is also based on combinator graph reduction (unlike Java). Graph reduction is designed specificly to deal with the issues of implementing functional languages with lazy semantics. Furthermore, the type system of Haskell is much richer than Java's, which makes verification more difficult. In particular, Java does not have first-class functions, a major feature of Haskell. The type system for Java bytecode also lacks parametric polymorphism.

Furthermore, Yhc bytecode differs from Java bytecode because it contains only forward branches. Thankfully, this difference simplifies the problem. With no backward branches, all bytecode control flow graphs will be acyclic. A major component of Java's bytecode verification system is Kildall's algorithm [4]. Kildall's algorithm is a fixed-point calculation over a control flow graph and has worst-case time complexity $O(n^2)$, where n is the length of the bytecode sequence [9]. This fixed-point calculation is required to deal with loops (backward branches) in Java bytecode. Because Yhc only has forward jumps, we do not need Kildall's algorithm but can instead rely on a linear time algorithms.

Starting in September 2005, the Yhc project has been 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 in C to execute the compiled programs.

NHC is also a bytecode compiler [37], 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, and debugging tools. However, 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 functional and implement the vast majority of the Haskell 98 standard. 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.

The present work shares some similarities with typed assembly language [26, 27]. However, work on TAL differs from this work 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 machine instruction sets. However, as some Haskell compilers do compile to native machine code, it might 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.

Bytecode verification is also related in spirit to proof-carrying code [29, 28]. It should be possible to recast the current work using the framework of PPC. However, the current formulation is easier to understand and implement than an alternate system utilizing the PPC framework, which employs a complete dependently-typed theorem verification engine. Because PPC is a very general framework, utilizing it requires a through understanding both of the PPC system itself and of the specific problem at hand. Nonetheless, relying on a well-tested, general-purpose theorem proof framework is also a strength. Now that the issues concerning typed G-Machine bytecode have been explored, it may be worthwhile to investigate reformulating the present work using the PPC framework.

Objective Caml is a widely-used variant of ML which has implementations of both bytecode and native machine code compilers. Although Meange sketches a design for an Objective Caml bytecode verifier, it has not, to our knowledge, been implemented [24]. The design so sketched is similar to the present work in the high level ideas. However, Objective Caml is a call-by-value language and thus its bytecode instruction set shares few similarities with the G-Machine. Another major difference is that the author suggests using unification during verification rather than using explicit type rewrites as we do. Relying on unification is problematic because unification in the presence of F_ω-style impredicative polymorphism is almost certainly undecidable.[28] Although compiling Haskell via translation to intermediate representations based on System F_ω is not strictly necessary, it is very convenient; working in a calculus with a weaker type system would make compiler construction considerably more difficult. In particular, our solution to the problem of typed labmda lifting would be impossible.

Govindavajhala and Appel have studied the possibility of circumventing security schemes based on static analysis (as ours is) by inducing memory errors in the hardware of the host machine [12]. If one has physical access to the host machine, it is possible to induce memory errors by, for example, subjecting the machine to heat outside its normal operating range. It is possible to construct programs which can take advantage of such memory errors and compromise the machine with high probability. Fortunately, such exploits can be avoided by using hardware-based solutions such as EEC memory.

[28] A large body of work exists which give undecidability results for even heavily-restricted versions of the second-order unification problem [11, 22, 38].

7 Future Work and Conclusions

In this paper we investigated bytecode verification for the Haskell language, specificly 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 defined the static and dynamic semantics of the IL using standard syntax-oriented inference rules. Although we have yet to work through the proofs, we conjecture that the IL possesses the important type-safety property.

Despite the original design goals of Haskell, the language does not have a standardized formal semantics. The Haskell Report informally references a simple "core" language, into which the more complicated language constructs can be translated [32]. Preliminary work was done to define the static [35] and dynamic [13] semantics of Haskell, but this work was never completed and formalized into a full semantics for the language.

In section 2, we introduced the IL used for this work. While discussing the various design decisions that went into the IL, we sketched a method for translating raw Haskell source into the IL. If this process were formalized, one could view the IL as the target of a denotational "compilation" semantics for Haskell. A possible avenue for future work is to investigate the possibility of using the IL as a stepping-stone for defining a full semantics for the Haskell 98 standard. Such work would validate one of the initial assumptions of our present work, which is that the IL is an appropriate intermediate language for the compilation of Haskell. Developing a denotational CPO semantics for the IL (as opposed to the operational semantics found here) would be an important part of any such work. The operational semantics given here over-specifies the evaluation order, and denotational semantics are conducive to equational reasoning on programs.

Another interesting avenue of investigation regarding the IL would be to prove and formally verify its meta-theory. Type-safety is the most important property we require. While the type system of the IL is complicated, it is built using well-understood extensions to the solid base of F_ω, and should be amenable to standard proof methods. Also, it would be valuable to directly prove the type safety of certified bytecode programs. The modal logic framework described in recent work by Appel may be a fruitful avenue of attack [6].

We also wrote a compiler which translates IL "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 IL (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.

Appendix – Full Yhc Bytecode Set Description

Note: much of the material in this appendix is derived from the official bytecode documentation contained in the Yhc source code. A human-readable version of this documentation can be viewed at `http://www-users.cs.york.ac.uk/~ndm/yhc/bytecodes.html`.

Bytecode instructions are classified into three groups: straight-line instructions, control-flow instructions and returning instructions. After executing a straight-line instruction the instruction pointer is always incremented by exactly one. Control flow instructions, however, may increment the instruction pointer by any legal amount and may cause the instruction pointer to be set to different values depending on the current state of the program. A returning instruction is one which ends the current stack frame when executed.

Within the straight-line instructions is a collection of instructions which perform basic operations on the Int, Float, and Double types. These are treated in a separate section from the other straight-line instructions.

Instructions which modify the local stack usually have accompanying stack-effect diagrams to help illustrate the effect. The top line of these diagrams represents the stack before the instruction is executed, and the bottom line represents the stack after. The top of the stack is oriented toward the left. Beyond the right edge of the digram one can imagine the bottom portion of the stack which is unaffected by the instruction. Each stack slot is described by a type and a status, as explained in section 5.1. In the effect diagrams, these are represented by a pair in each cell; the first component of the pair is the type and the second is the status. The status codes are:

- N – Normal
- E – Evaluated
- U – Uninitialized
- Z – Zapped

Either component of a stack slot may contain a variable instead of a concrete type or status. If the variable appears in both lines, the corresponding slots are asserted to have the same type or status. If a concrete type or status appears in the top line, it is a precondition of the instruction that the corresponding stack slot have that type or status.

A frequent precondition of bytecode instructions is that some stack position be non-zapped. This is indicated in the top line of the diagram with the statement $s \neq Z$, for some status variable s.

Indices into the stack or argument list always start at 0.

Straight-line Instructions

`END_CODE`

This "instruction" is actually illegal. It is placed at the end of each combinator bytecode sequence as a safeguard against buggy compilers. If this instruction is executed, the runtime will immediately exit with an error. For verification purposes, this instruction is treated as an error if it is encountered along any control flow path.

`PRIMITIVE`

This instruction is used to execute runtime primitives and to invoke the foreign function interface (FFI). See the Yhc documentation for details on how this instruction works.

For the purposes of verification, this instruction is considered an error wherever it appears. It is impossible to verify the type correctness of uses of (potentially unsafe) runtime primitives and foreign functions. We therefore cannot allow their use in untrusted code.

It might instead be possible to define a subset of runtime primitives which are safe and make those primitives available via this instruction. For this to work, the meaning and type effects of this set of primitives would have to be standardized. We leave this for future work.

`NEED_HEAP` x

This instruction is used to reserve heap space for the instructions which follow it. When executed, this instruction checks to be sure that at least $32x$ words are available in the heap. If there is not enough space in the heap, garbage collection is performed. For verification, this instruction increases the amount of reserved heap by $32x$.

`PUSH` n

Pushes the nth item on the stack onto the top.

If $s_n = U$, then $s'_n = N$. Otherwise, $s'_n = s_n$.

	t_0, s_0	$\cdots$	$t_n, s_n \neq Z$
t_n, s'_n	t_0, s_0	$\cdots$	t_n, s_n

`PUSH_ZAP` n

Pushes the nth item on the stack onto the top and additionally zaps the nth position on the stack.

	t_0, s_0	$\cdots$	$t_n, s_n \neq Z$
t_n, s_n	t_0, s_0	$\cdots$	t_n, Z

`ZAP_STACK` n

Zap the nth position on the stack.

t_0, s_0	$\cdots$	$t_n, s_n \neq Z$
t_0, s_0	$\cdots$	t_n, Z

`PUSH_ARG` n

Push the nth argument onto the top of the stack. The argument must not be zapped. Let a_n be the type of the nth argument.

a_n, N

PUSH_ZAP_ARG n

Push the nth argument onto the top of the stack. The argument must not be zapped. Additionally, zap the nth argument. Let a_n be the type of the nth argument.

a_n, N

ZAP_ARG n

Zap the nth argument. The argument must not be already zapped.

PUSH_INT x

Push the literal, signed integer value x onto the stack.

$@\{\!\mid \mathit{Int} \mid\!\}, E$

PUSH_CHAR x

Push the literal, unsigned integer value x onto the stack. The Yhc runtime does not have a separate character primitive type; characters are treated as integers.

$@\{\!\mid \mathit{Int} \mid\!\}, E$

PUSH_CONST n

Push the nth constant value from the constant table onto the stack. The type of the constant depends on the entry in the constant table. See the Yhc documentation for details.

t, E

MK_AP n

Create a fully saturated application node using the nth entry in the constant table, which must be a function info entry. Let t be the type of the indicated function after polymorphic rewrites are applied. Then, we must have that

$$t = t_0 \to \cdots \to t_{m-1} \to t'$$

where m is the arity of the combinator.

$t_0, s_0 \neq Z$	$\cdots$	$t_{m-1}, s_{m-1} \neq Z$
		t', N

`MK_PAP` n m

Make a partial application using the nth entry in the constant table, which must be a function info entry. The arity of the indicated function must be $\geq m$. Let t be the type of the function after polymorphic rewrites are applied. Then, we must have that

$$t = t_0 \rightarrow \cdots \rightarrow t_{m-1} \rightarrow t'$$

If m is less than the arity of the indicated function, then the status $s' = E$. Otherwise $s' = N$.

$t_0, s_0 \neq Z$	$\cdots$	$t_{m-1}, s_{m-1} \neq Z$
		t', s'

`APPLY` n

Apply the function on the top of the stack to n additional arguments from the stack. If applying an extra n arguments to the application a would super-saturate it (i.e. apply it to more arguments that the function's arity) then `APPLY` saturates the application fully and then builds further applications to the built-in function `_apply` to apply the rest of the arguments.

The function `_apply` is defined as the bytecode sequence:

```
NEED_HEAP 1
PUSH_ZAP_ARG 1
PUSH_ZAP_ARG 0
EVAL
APPLY 1
RETURN_EVAL
```

which is to say it evaluates the fully-saturated application which then returns another application, and this application is then applied to the additional argument.

See the Yhc documentation for more details.

Let t' be the type t after polymorphic rewrites are applied. Then we must have

$$t' = t_0 \rightarrow \cdots \rightarrow t_{n-1} \rightarrow t''$$

$t, s \neq Z$	$t_0, s_0 \neq Z$	$\cdots$	$t_{n-1}, s_{n-1} \neq Z$
			t'', N

`MK_CON` n

Build a constructor node using the nth item of the constant table, which must be a constructor info entry. Let m be the arity of the product payload of the data constructor. Let i be the tag value of the data constructor.

$t_0, s_0 \neq Z$	$\cdots$	$t_{m-1}, s_{m-1} \neq Z$
		$\{\!\mid i : \langle\!\mid t_0, \cdots, t_{m-1} \mid\!\rangle \mid\!\}, E$

`UNPACK`

Take a data constructor with a known tag and unpack the contents of its data payload.

		$\{\!\!\mid i : \langle\!\!\mid t_0, \cdots, t_{m-1} \mid\!\!\rangle \mid\!\!\}, E$
t_0, N	$\cdots$	t_{m-1}, N

`SLIDE` n

Temporarily remove the top item of the stack, pop the next n items on the stack, and then replace the original item.

$t, s \neq Z$	t_0, s_0	$\cdots$	t_{n-1}, s_{n-1}
			t, s

`POP` n

Pop the top n items off of the stack.

t_0, s_0	$\cdots$	t_{n-1}, s_{n-1}

`ALLOC` n

Generate n place-holder heap nodes. Initially, these nodes will abort the program with an error if evaluated. They are intended to be overwritten with the `UPDATE` instruction. Types are assigned to the place-holder nodes via the polymorphic rewrite rules; there must be exactly n polymorphic rewrite rules for this instruction. Let t_0 through t_{n-1} represent these types.

t_0, U	$\cdots$	t_{n-1}, U

`UPDATE` n

Remove the top item from the stack. Then, overwrite the nth item (in the remaining stack) with the removed item. We must have $t = t_n$.

$t, s \neq Z$	t_0, s_0	$\cdots$	t_n, U
	t_0, s_0	$\cdots$	t, s

`EVAL`

Evaluate the top item on the stack to weak head normal form. This instruction has no runtime effect if the item is an unsaturated application or if the item is a constructor.

$t, s \neq Z$
t, E

`SELECTOR_EVAL`

This instruction is an abbreviation for the bytecode sequence:

```
PUSH_ARG 0
EVAL
```

This sequence occurs frequently due to the translation of type-class dictionaries and records.

a_0, E

`STRING`

Convert a primitive string value into a Haskell string value, which is represented by a list of characters.

See the Yhc documentation for details on how this instruction works.

$@\{\!\mid CString \mid\!\}, E$
$\$\textsc{Prelude.String}, E$

`FROM_ENUM`

Remove the top item on the stack (which must be a data constructor), and push an integer containing the tag number of the constructor.

$\{\!\mid i_0 : t_0, \cdots, i_{m-1} : t_{m-1} \mid\!\}, E$
$@\{\!\mid Int \mid\!\}, E$

Control-flow Instructions

The instructions in this section are instructions which cause branching. Some additionally have stack effects.

`JUMP` x

Unconditionally increment the instruction pointer by x bytes. For verification, this instruction copies its stack state to its jump location.

`JUMP_FALSE` x

Remove the top item from the stack (which must be a boolean value). If it is false, increment the instruction pointer by x bytes. Otherwise, continue execution at the next instruction.

For verification, this instruction has the following stack effect, which affects both the immediately following instruction and its jump target.

$\$\textsc{Prelude.Bool}, E$

`INT_SWITCH` $x\ j\ js$

This instruction takes three arguments. The first is the number of alternatives in the switch. The second argument is the default brach jump location. The third argument is a list of signed integer, jump location pairs.

When executed, this instruction examines the top item on the stack, which must be an Int. If the integer matches any of the values in the jump list, the instruction pointer is incremented by the given amount. If no value matches, the default branch is taken.

The following stack effect applies to all branches (including the default branch).

$$\begin{array}{|c|}\hline @\{\!|\ \mathit{Int}\ |\!\}, E \\ \hline @\{\!|\ \mathit{Int}\ |\!\}, E \\ \hline \end{array}$$

`LOOKUP_SWITCH` $x\ j\ js$

This instruction takes three arguments. The first is the number of alternatives in the switch. The second argument is the default brach jump location. The third argument is a list of unsigned integer, jump location pairs.

When executed, this instruction examines the top item on the stack, which must be a data constructor. If the tag value matches any of the values in the jump list, the instruction pointer is incremented by the given amount. If no tag value matches, the default branch is taken.

For verification, we require that the type of the top item on the stack be a sum type and that every tag value that appears in the jump list also appear in the sum.

The following stack effect applies to the default branch:

$$\begin{array}{|c|}\hline \{\!|\ i_0 : t_0, \cdots, i_{n-1} : t_{n-1}\ |\!\}, E \\ \hline \{\!|\ i_0 : t_0, \cdots, i_{n-1} : t_{n-1}\ |\!\}, E \\ \hline \end{array}$$

The following stack effect applies to a branch corresponding to the data constructor matching the value i_j:

$$\begin{array}{|c|}\hline \{\!|\ i_0 : t_0, \cdots, i_{n-1} : t_{n-1}\ |\!\}, E \\ \hline \{\!|\ i_j : t_j\ |\!\}, E \\ \hline \end{array}$$

`TABLE_SWITCH` $x\ js$

This instruction takes two arguments. The first is the number of switch alternatives. The second is a list of jump locations.

This instruction is very similar to the `LOOKUP_SWITCH` instruction, except that the jump list contains only jump locations and not tag value, jump location pairs. For `TABLE_SWITCH`, each jump location has an assumed tag value equal to its position in the list. `TABLE_SWITCH` also has no default branch.

When executed, this instruction examines the top item on the stack, which must be a data constructor. The instruction pointer is then incremented by the amount of the ith element in the jump list, where i is the tag value of the constructor.

For verification, we require that the type of the top item on the stack be a sum type. We also require that the sum type have exactly the tag values from 0 to $x - 1$.

The following stack effect applies to a branch corresponding to the data constructor matching the value i:

$\{\!\!\mid 0 : t_0, \cdots, x-1 : t_{x-1} \mid\!\!\}, E$
$\{\!\!\mid i : t_i \mid\!\!\}, E$

Returning Instructions

These instructions cause the stack frame to return. Therefore, all the instructions in this section have no stack effects. However, they all have preconditions which ensure that the combinator returns a data item of the correct type.

RETURN

This instruction returns from the current stack frame using the top value on the local stack as the return value.

For verification, the top item on the stack must have the correct type for the return value and it must be in state "Evaluated."

RETURN_EVAL

This instruction is an abbreviation for the following bytecode sequence:

```
EVAL
RETURN
```

except that it also implements tail-call elimination. Operationally, the current stack frame is popped before the new stack frame is created.

For verification, the top item on the stack must have the correct type for the return value and it may be in any non-zapped state.

SELECT n

This instruction is an abbreviation for the following bytecode sequence:

```
UNPACK
PUSH_ZAP n
RETURN_EVAL
```

This sequence appears frequently when implementing type-class dictionaries and record selectors.

For verification, the top item on the stack must be a sum type with exactly one variant, and the nth element of the product must have the correct return type.

Numeric Primitive Instructions

There are primitive operations dealing with three of the basic numeric types in Haskell: Int (bounded integers), Float (single-precision floating-point), and Double (double-precision floating-point).

These instructions are rather arbitrarily chosen from the set of primitive operations required by the Haskell 98 standard. Operations not found here are implemented in the base library using the FFI to call the appropriate C functions.

`ADD_W`, `SUB_W`, `MUL_W`, `DIV_W`, `MOD_W`

Each of these instructions performs one of the five basic numeric operations on integers. Respectively, these are: addition, subtraction, multiplication, integer division, integer modulus. Each instruction has the same stack effect.

@⦃ *Int* ⦄, *E*	@⦃ *Int* ⦄, *E*
	@⦃ *Int* ⦄, *E*

`EQ_W`, `NE_W`, `LE_W`, `LT_W`, `GE_W`, `GT_W`

Each of these instructions performs a comparison operation on integers. Respectively, these are: equality, inequality, less-than-or-equal, less-than, greater-than-or-equal, greater-than. Each instruction has the same stack effect.

@⦃ *Int* ⦄, *E*	@⦃ *Int* ⦄, *E*
	$Prelude.Bool, *E*

`NEG_W`

Unary negation of bounded integers.

@⦃ *Int* ⦄, *E*
@⦃ *Int* ⦄, *E*

`ADD_F`, `SUB_F`, `MUL_F`, `DIV_F`

Each of these instructions performs one of the four basic numeric operations on single-precision floating-point values. Respectively, these are: addition, subtraction, multiplication, and division. Each instruction has the same stack effect.

@⦃ *Float* ⦄, *E*	@⦃ *Float* ⦄, *E*
	@⦃ *Float* ⦄, *E*

`MOD_F`

This instruction indicates the modulus of a floating-point number. It is nonsense and causes an error if executed. For verification purposes, it is considered an error wherever it appears.

`EQ_F`, `NE_F`, `LE_F`, `LT_F`, `GE_F`, `GT_F`

Each of these instructions performs a comparison operation on single-precision floating-point values. Respectively, these are: equality, inequality, less-than-or-equal, less-than, greater-than-or-equal, greater-than. Each instruction has the same stack effect.

@⦃ *Float* ⦄, *E*	@⦃ *Float* ⦄, *E*
	$Prelude.Bool, *E*

NEG_F

Unary negation of single-precision floating-point values.

$@\{\!\!\mid \mathit{Float} \mid\!\!\}, E$
$@\{\!\!\mid \mathit{Float} \mid\!\!\}, E$

ADD_D, SUB_D, MUL_D, DIV_D

Each of these instructions performs one of the four basic numeric operations on double-precision floating-point values. Respectively, these are: addition, subtraction, multiplication, and division. Each instruction has the same stack effect.

$@\{\!\!\mid \mathit{Double} \mid\!\!\}, E$	$@\{\!\!\mid \mathit{Double} \mid\!\!\}, E$
	$@\{\!\!\mid \mathit{Double} \mid\!\!\}, E$

MOD_D

This instruction indicates the modulus of a floating-point number. It is nonsense and causes an error if executed. For verification purposes, it is considered an error wherever it appears.

EQ_D, NE_D, LE_D, LT_D, GE_D, GT_D

Each of these instructions performs a comparison operation on double-precision floating-point values. Respectively, these are: equality, inequality, less-than-or-equal, less-than, greater-than-or-equal, greater-than. Each instruction has the same stack effect.

$@\{\!\!\mid \mathit{Double} \mid\!\!\}, E$	$@\{\!\!\mid \mathit{Double} \mid\!\!\}, E$
	$\$\textsc{Prelude.Bool}, E$

NEG_D

Unary negation of double-precision floating-point values.

$@\{\!\!\mid \mathit{Double} \mid\!\!\}, E$
$@\{\!\!\mid \mathit{Double} \mid\!\!\}, E$

References

[1] Lennart Augustsson. Implementing haskell overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.

[2] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422, pages 52–67. Springer-Verlag, Berlin, 1998.

[3] Richard S. Bird and Ross Paterson. de bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999.

[4] Solange Coupet-Grimal and William Delobel. A Uniform and Certified Approach for Two Static Analyses. Research report 24-2005, LIF, Marseille, France, April 2005.

[5] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to Netscape and beyond. In IEEE, editor, *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.

[6] Andrew Appel et al. A very modal model of a modern, major, general type system. In *POPL '07, Proceedings*, 2007. to appear.

[7] Stephen N. Freund and John C. Mitchell. A formal framework for the java bytecode language and verifier. *ACM SIG-PLAN Notices*, 34:147–166, 1999.

[8] Stephen N. Freund and John C. Mitchell. A type system of object initialization in the java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21:1196–1250, 1999.

[9] Andreas Gal, Christian W. Probst, and Michael Franz. A Denial of Service Attack on the Java Bytecode Verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, November 2003.

[10] 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.

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

[12] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *2003 IEEE Symposium on Security and Privacy: May 11–14, 2003.* IEEE Computer Society Press, 2003.

[13] Kevin Hammond and Cordelia Hall. A dynamic semantics for haskell (draft).

[14] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 341–354, New York, NY, USA, 1990. ACM Press.

[15] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10, New York, NY, USA, 1982. ACM Press.

[16] R. J. M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Programming Research Group, Oxford University, July 1983.

[17] Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 99–110, New York, NY, USA, 2004. ACM Press.

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

[19] Gerwin Klein. *Verified Java Bytecode Verification.* PhD thesis, Technische Universität München, 2003.

[20] 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.

[21] 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.

[22] Jordi Levy. Decidable and undecidable second-order unification problems. *Lecture Notes in Computer Science*, 1379, 1998.

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

[24] Paul B. Meange. Resource control of untrusted code in an open network environment. Technical Report UCAM-CL-TR-561, University of Cambridge, 2003.

[25] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions (summary). In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 308–319, New York, NY, USA, 1986. ACM Press.

[26] 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.

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

[28] G. Necula and P. Lee. Research on proof-carrying code on mobile-code security. In *Proceedings of the Workshop on Foundations of Mobile Code Security*, 1997.

[29] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[30] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[31] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[32] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

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

[34] S. Peyton-Jones and E. Meijer. Henk: a typed intermediate language. In *Proceedings of the Types in Compilation Workshop*, June 1997.

[35] S. Peyton-Jones and Philip Wadler. A static semantics for haskell. Technical report, 1992.

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

[37] 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.

[38] Aleksy Schubert. Second-order unification and type inference for church-style polymorphism. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 279–288, New York, NY, USA, 1998. ACM Press.

[39] Rayme Stata and Marin Adabi. A type system for java bytecode subroutines. In *25th Symposium on the Principles of Programming Languages*, pages 148–160, 1998.

[40] J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.