Types and Intermediate Representations

Michael Hicks
University of Pennsylvania

The design objectives and the mechanisms for achieving those objectives are considered for each of three systems, Java, Erlang, and TIL. In particular, I examine the use of types and intermediate representations in the system implementation. In addition, the systems are compared to examine how one system's mechanisms may (or may not) be applied to another.

---

1. INTRODUCTION

Considerable research has been done about the use of *types* in high-level programming languages. Types assist in describing a program, so that it may be more easily understood by a human or a compiler. They provide representation information to the compiler, such as whether to store a variable in a general purpose or a floating point register. They reduce the occurrence of malformed programs by preventing type-incorrect operations, for example the addition of an integer and a string. They may help guarantee properties about a program, such as whether or not it will terminate. In all of these capacities, types serve to strengthen and simplify the language and its implementation.

Another useful tool to language designers/implementors is the use of *intermediate representations*. Frequently, a compiler does not translate a program in a high-level language directly to machine code, but instead first translates it to a program in a slightly smaller, simpler language. This new program is then mapped to a program in an even simpler language, and so on, until the end result is reached. These simpler versions of the original are intermediate representations of the program. Types may also be applied to these simpler intermediate representations with the same benefits mentioned above, further strengthening the implementation.

This paper explores the use of types and intermediate representations in the design and implementation of three languages: Java, Erlang, and Standard ML (SML). In Java and Erlang, source programs are compiled to be understood by a virtual (or abstract) machine; for Java this is called the JVM (Java Virtual Machine) [Lindholm and Yellin 1995], while for Erlang a number of abstract machine

---

This paper was written to satisfy the **Written Preliminary Exam II** prerequisite for obtaining a degree of Doctor of Philosophy in Computer and Information Science at the University of Pennsylvania.

Committee:
Carl Gunter, chair
Scott Nettles
Peter Buneman

The assigned works of this paper are [Morrisett 1995], chapter 8, [Lindholm and Yellin 1995], Sections 4.8 and 4.9, and various works about the Erlang abstract machine (the core being [Armstrong et al. 1992; Hausman 1994]). All other references served to bolster information presented in these works.

implementations exist. I shall look at the JAM (Joe's Abstract Machine) [Armstrong et al. 1992], and the BEAM (Bogdans's Erlang Abstract Machine) [Hausman 1994]. For SML, I look at the TIL [Morrisett 1995; Tarditi 1996] compiler, specifically, its use of typed intermediate representations as it compiles to Alpha assembly code.

The presentation of the paper is as follows. I first present an informal background on the uses and terminology of types. This is followed by a discussion of each of the three systems. I will first spend a brief time introducing the language, its features and constructs, and the design goals of the language system. I then explain how these goals are achieved, highlighting the use types and intermediate representation(s), and to what advantages and disadvantages. I present any pertinent implementation details and performance measurements of the system. I follow these three discourses with a general discussion relating the uses of types in the systems to each other, and how one system might benefit from the techniques developed in another. I end with conclusions and related work.

Throughout this paper, the word *expression* or *term* will be used to refer to a fragment of a *program*. A program is defined by the nature of the programming language, and will be covered in the respective language sections. The *evaluation* of a program or expression serves to reduce the program to a base form, called a *value*; the method of reduction is defined by the semantics of the programming language.

2. TYPES

We begin with a brief discussion of programming language *types*. While it is expected that the reader at least has a vague understanding of these concepts, this treatise shall serve to more concretely define some of the terminology used in the rest of the paper. This discussion relies on informal language; a more formal treatment of types may be found in [Gunter 1992]. In this paper we shall look at two uses of types by the compiler: as an indicator of well-formed expressions, and as a hint for determining the representation of values at runtime. A number of sophisticated types will be examined in these contexts.

2.1 Well-formed Expressions

According to Gunter [1992], "A *type* can be viewed as a property of an expression that describes how the expression can be used." Commonly, a type represents a set of legal values, such as the integers, and an expression of that type is expected to represent member of that set. Therefore, an expression of integer type should be allowed wherever an integer is allowed, such as an argument to the function `+`.

A program P is *type-correct* if there is no stage in the evaluation of P that causes an expression to be used in a context not allowed by its type. For example, a program that attempted to perform the operation

```
"a" + 7
```

would not be type correct (in most languages), because the function `+` expects both of its arguments to have numerical type, but in this expression, the first argument has type `string`.

2.1.1 *Type checking.* The process of determining the type-correctness of a program is called *type-checking*. One question is: when is type-checking performed? Programming languages with *static type-checking* (or simply, *static typing*) can determine this type-correctness property at *compile-time*, that is, without actually evaluating the program. SML and C[1] are statically typed. On the other hand, languages with *dynamic type-checking* (or simply, *dynamic typing*), such as Scheme and Erlang, wait to determine type-correctness during the evaluation of the program. Of course, a hybrid approach may be taken as well, as is the case of Java (further explained in Section 4).

Statically typed languages make use of the property of *type soundness*: if an expression M has type t and M evaluates to value V, then V also has type t. In other words, the type of an expression will not change as it is evaluated. If this property did not hold for the type-system of the language, then static checks would not be sufficient for determining type-correctness.

Furthermore, static type-checking should be *decidable*, that is, the process of type-checking should always terminate (with success or failure) for a syntactically correct program, regardless of whether the program itself terminates.

Dynamically-typed languages require that the types of values be present at runtime. For instance, the function `+` needs to verify that its arguments have numerical type at runtime; this implies that, for the above example, the types of `"a"` and `7` need to be available at runtime. As a result, dynamically-typed languages often make *type predicates* available to the programmer: type predicates allow the type of a value to be tested and recognized at runtime. For example, Scheme has type predicates `boolean?`, `pair?`, `symbol?`, `number?`, `char?`, `string?`, `vector?`, and `procedure?` [Clinger and Rees 1991]. Such predicates are usually not present in statically-typed languages because type information may not available at runtime[2].

What are the advantages and disadvantages of these approaches? Generally, static typing trades expressiveness for safety and speed. This is because, as mentioned, some programs may be compiled without runtime type information, saving space and runtime checks. In addition, type-errors are caught at the outset, rather than cropping up in some boundary case after system deployment. On the other hand, the type system may fail to correctly type-check a number of valid[3] programs. For example, this perfectly legal Scheme program fails type-checking in SML because it may return values of different types (either a string or an integer)[4], which is forbidden by the SML type system:

```
(define (f x)
  (if (> x 1)
    "hello"
    2))
```

[1]However, C is not *strongly* typed; see the next section.
[2]This is certainly the case with C, which has no use for runtime type information. With SML, some information is needed to enable *polymorphism* and *garbage collection*.
[3]Valid here means programs that will terminate without error, when used in the proper context.
[4]Another, perhaps more compelling example may be found in [Gunter 1992], pages 221-223

2.2 Representation and Strong Typing

In programming language implementations, types help the compiler determine how program values are to be represented at runtime. Bit patterns in memory or registers serve to *encode* higher-level constructs such as integers, floating point values, pictures, etc. Sometimes the encoding must be recognized by the architecture, as is the case with integers and floating point numbers (since they need to be understood by the integer and floating point ALU's), which thus dictates such issues as register allocation (an integer must go into a general purpose register, and not into a floating point one).

Tightly coupled with the policy of encoding is the characteristic of *strong typing*. A programming language is strongly typed if its type system cannot be compromised. To understand this, consider the following example. Most programming languages support the concept of an *enumerated type*, which is simply a list of names that form a set. In C:

```
enum color {red,blue,green,purple};
```

This statement defines a type `color` which is a set of four elements named `red`, `blue`, and `green` and `purple`. A runtime value of type `color` must be encoded in some bit-pattern on the executing machine. One possible encoding could allocate 8 bits per element as follows:

```
red    = 00000000
blue   = 00000001
green  = 00000010
purple = 00000011
```

Note that these bit values also represent an encoding of the integer values `1`, `2`, `3`, and `4`, respectively. Therefore, while in the "real-world", it doesn't make sense to add a color to a number, we might feasibly add `1` and the *encoding* of `blue` and end up with `2`, as in the following piece of C code:

```
...
color b = blue;
int x = 1 + (int)blue;
printf("%d",x);
...
```

which would output `2`. The key operation here is a *cast* (the `(int)blue` part) which allows the program to treat `blue` *as if it were an integer*, thereby revealing the encoding. Casts can compromise safety when used to construct arbitrary pointers into memory, as in:

```
int a = 1001;
int *ptr_to_a = &a;                    /* get pointer to a; OK */
int *ptr_to_something = (int *)1001;  /* create pointer using
                                          encoding of 1001 */
*ptr_to_something = 11;                /* set value at pointer to 11;
                                          could be disastrous */
```

SML and Scheme do not permit such casts, and Java allows only a type-safe form of them (which doesn't include encoding arbitrary pointers). Such casts essentially compromise the type system because they allow values to be used in contexts in which they weren't intended.

Strongly typed languages are safer but perhaps less expressive. This is in part why C is such a popular language: its lack of strong typing allows programs to more directly access to the underlying representation for more straightforward manipulations. This can be especially useful in the latter stages of a compiler, or in a garbage collector. On the other hand, programs that subjugate the type system are more buggy and harder to read. We shall see in Section 3 that TIL proves a compiler *can* make use of strong typing without great loss of flexibility.

2.3 Advanced Types

In addition to the basic, set-based types available in languages like C, some languages provide more expressive types.

2.3.1 *Parametric Polymorphism.* Often, we would like to write functions that operate on the structure of data without being concerned with particular values in that structure. For example, the Scheme function `swap`, indicated below, swaps the first and second elements of a pair:

```
(define (swap pair)
  (cons (cdr pair) (car pair))) ;new pair with elements swapped
```

Never are the values of the elements of the pair consulted. Thus, this function should work equally well for pairs of integers, pairs of lists, etc.

In statically-typed languages, the compiler/programmer is faced with the problem of assigning a type for such a function at compile time. To do it, a static type system may employ *parametric polymorphism* which allows functions to be applicable to a family of types. In ML, we can write a function `swap` that swaps the elements of a tuple:

```
fun swap (a,b) = (b,a)
```

This term has the type $\alpha \times \beta \rightarrow \beta \times \alpha$. This means that `swap` is a function that may be applied to a pair whose first element has type α and whose second element has type β, and it will return a pair whose first element has type β and whose second element has type α. α and β are *type variables*, whose value is filled in at runtime by context.

The presence of polymorphism makes compilation difficult. To understand why, consider how the compiler is to allocate space for the polymorphic arguments of the `swap` function. These arguments could be simple types, like integers, but they also might be more complicated structures, so the amount of space required at runtime unknown at compile-time. We shall see in TIL's approach to this problem in Section 3.

2.3.2 *Recursive and Variant Types.* Variant types are those that may have multiple forms. The enumerated type for `color`, in the above example, is a form of variant type. In SML, `color` could be declared:

```
datatype color = Red | Blue | Green | Purple;
```

Each of the capitalized names on the right-hand side of the = is called a *constructor*, because they serve to construct the type `color`. In SML, constructors may also take arguments. For example:

```
datatype thing = Human of string | Dog of string;
```

In the context of a program, values of type `thing` will appear with a constructor followed by its argument. For example, `Human("alice")`, `Human("bob")`, and `Dog("fido")` all define values having type `thing`. The constructor serves to distinguish which variation of the type is in use for a particular value; in this example, the constructors determine things are humans, and which are dogs.

Recursive types are those that may be defined in terms of themselves. For example:

```
datatype intlist = Cons of (int * intlist) | Nil
```

This serves to define lists of integers. This declaration is recursive, as `intlist` appears on both the right and left-hand sides. The `Cons` constructor serves to tag elements that are in the list, and `Nil` serves as the list terminator. For example, `Cons(1,Cons(2,Cons(3,Nil)))`, `Nil`, and `Cons(6,Nil)` all have type `intlist`. Recursive types are very powerful, as they easily facilitate the use recursive functions as De-constructors of the data.

A longer treatise of both of these topics in the context of SML may be found in [Myers et al. 1994], pages 153-172.

3. TIL

I shall begin the body of the paper by describing TIL, a compiler developed largely by Greg Morrisett and David Tarditi at Carnegie Mellon University for the language Standard ML. TIL has the most sound theoretical foundations of all of the systems that I examine in this paper.

Standard ML is a "mostly-functional" programming language that originated in the early 70's for uses in automated theorem-proving. Its form is based on the simply-typed lambda calculus, so it is widely used in the research community. Since its origin, SML's popularity has grown for use in general-purpose applications, and a number of highly-tuned compilers and standard libraries have emerged [Appel and MacQueen 1987; Leroy 1995; Berry 1991].

3.1 Language Features

SML encourages a functional, or *declarative*, style of programming; this means that SML programs describe solutions to a problem, whereas *imperative* languages provide instructions to a computer. As a result, declarative languages have proven more of a challenge to compile since they are less related to the underlying machine; however, it is typically easier to develop correct, understandable programs [Armstrong et al. 1992].

SML programs consist of a series of function and data definitions which interoperate at runtime. One of the features of SML is that functions may be *higher-order*, that is, functions may be constructed "on the fly", bound to variables, passed as arguments to other functions and returned as values. When an item may be manipulated as data, it is called *first class*; therefore, it is also the case that SML has first-class functions.

In addition, SML programmers have access to a number of built-in datastructures, including uniformly-typed, variable-length *lists*, fixed-length *tuples* of distinctly-typed values, arrays, and others. Some other features of SML include a module system, first-class exceptions, recursive user-defined datatypes, and pattern matching.

A suitable description of SML may be found in [Myers et al. 1994].

3.1.1 *The Type System.* SML terms are strongly typed, and type-checked statically. In addition, SML is able to *infer* the types of most terms at compile-time, thereby mitigating the need for programmer-supplied type labels. ML allows for the creation of user-defined variant types, using the `datatype` construct; these types may be recursive. SML also supports parametric polymorphism.

3.2 Goals

TIL was designed with a number of goals in mind. They are[5]:

—**Make the common case fast**, possibly at the expense of a less common one. This implies that the cost of the advanced features of SML, such as polymorphism, should only be incurred by programs that use those features.

—**Propagate type information** through as many stages of the compilation as possible. The idea was to try to discover ways in which types could be used to aid the lower levels of compilation, such as during the optimization phases.

—**Leverage existing tools**. Due to specialized implementations, many SML compilers (notably SML/NJ [Appel and MacQueen 1987]) cannot take advantage of standard tools, such as profilers, debuggers, and the standard linking facility `ld`. TIL attempts to rectify this situation so that it integrates more easily into the standard UNIX programming environment.

—**Interoperability**. SML implementations are notoriously bad at interoperating with other languages, in particular, C. The designers wanted to make this interface as straightforward as possible to allow access to standard libraries and hardware, thus enabling ML "systems" programming [Harper and Lee 1994; Alexander et al. 1997].

I will first motivate TIL's implementation by a discussion of the difficulties of compiling SML in the face of the above goals. Much of this difficulty comes from having to compile polymorphism. This design discussion is then followed by more implementation details.

3.3 The Design

The presence of polymorphic types increases the complexity of compilation. For the most part, this is because part of the type may be unknown at compile-time, and variable at runtime. Recalling the example of the `swap` function from Section 2.3.1, consider the use of types by the compiler for representation: how much space should be allocated for the input pair in the compiled function? There are many

[5] see [Morrisett 1995], Section 8.1 for more detail.

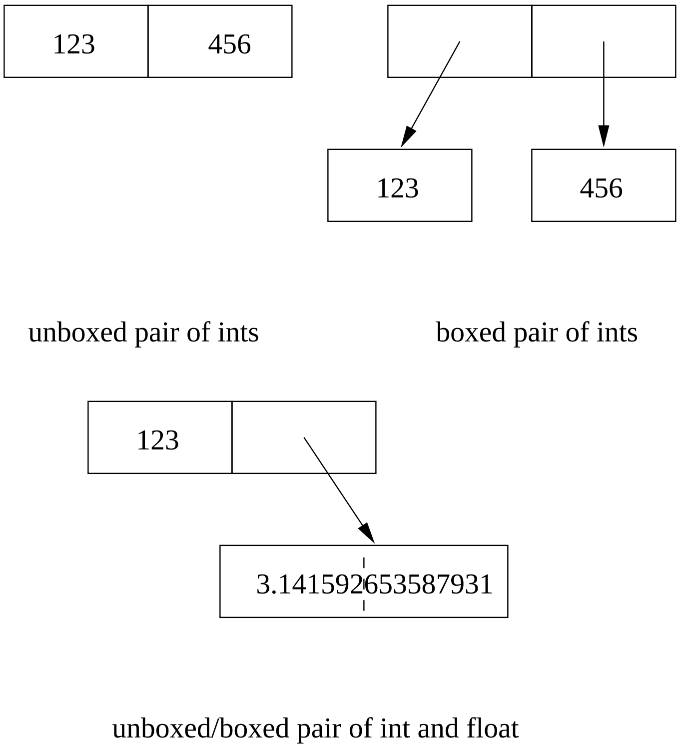

Fig. 1. Comparing boxed and unboxed representations

possibilities, since the type does not provide any constraint. For example, if the members of the pair were floating point numbers, 64 bits would be required for each, while if they were integers, 32 bits each would be required. One approach is to force *all* types to be of uniform size, say, 1 machine word. This is facilitated by the use of *boxed* representations: instead of storing the elements directly "in the pair", pointers to the elements are stored instead. This is illustrated in Figure 1. The figure also illustrates that the boxing scheme can be optimized by only indirecting references to objects that are larger than a single word.

Polymorphic data also makes it difficult to implement garbage collection. Garbage collection serves to reclaim memory that is no longer in use so that it may be reallocated. This is often accomplished by tracing: a set of *roots* that are directly accessible by the running program are identified (such as values in the stack and machine registers), and then all data *reachable* (directly or transitively) via pointers from those roots is preserved, while the rest is reclaimed for future allocation. This requires that at runtime the garbage collector be able to distinguish which words are pointers and which are not, so that traversal may proceed correctly. For monomorphic languages, data representation is known at compile time, and could thus be made available to the collector at runtime. However, when polymorphic data is present, this representation may not be available during compilation. One approach is to *tag* each word as a pointer or a non-pointer when an object is allocated at runtime. This is usually implemented by reserving some number of bits of each word to represent the tag; in the best case only 1 bit is needed, and the remaining bits serve to represent the value. SML/NJ uses both boxing and tagging.

These approaches have a number drawbacks. First, an extra space overhead is introduced for each occurrence of boxing, which is coupled with a time overhead since accesses require an additional indirection. Second, tagging reduces precision; in our example, integers may now only be 31 bits, as the remaining bit is used for the tag. The tag also inhibits direct interaction with foreign language functions (such as C functions) that use an untagged encoding – some representation conversion must take place before and after the call to the foreign function. Finally, it should be noted that this overhead is present even in code that doesn't use polymorphism.

These drawbacks contradict the goals of TIL. For example, more common monomorphic code is slowed in order to enable polymorphism, and interoperability is compromised by tagging. Therefore, an approach is needed that rectifies these problems and also supports the other goals of TIL: namely, to propagate type information and leverage standard tools.

3.3.1 *Dynamic Type Dispatch.* A cornerstone in the design of TIL that enables the compilation of polymorphic code is a technique called *dynamic type dispatch*. The idea is to dispatch specialized pieces of code based on the type of the values being used. As an example (given in [Morrisett 1995]), consider the polymorphic array subscript function sub which has the SML type α $\mathsf{array} \times \mathsf{int} \rightarrow \alpha$[6]. In TIL, this function is first compiled to an intermediate form that supports type dispatch, given as:

$$\begin{aligned}\texttt{sub = } \Lambda\alpha.\ & \texttt{typecase } \alpha \texttt{ of} \\ & \quad \mathsf{int} \texttt{ => intsub} \\ & \mid \mathsf{float} \texttt{ => floatsub} \\ & \mid \mathsf{ptr}[\sigma] \texttt{ => ptrsub}[\sigma]\end{aligned}$$

which assumes the following operations:

$$\begin{aligned}\texttt{intsub} &\ :\ \mathsf{intarray} \times \mathsf{int} \rightarrow \mathsf{int} \\ \texttt{floatsub} &\ :\ \mathsf{floatarray} \times \mathsf{int} \rightarrow \mathsf{float} \\ \texttt{ptrsub}[\mathsf{ptr}[\sigma]] &\ :\ \mathsf{ptrarray}[\mathsf{ptr}[\sigma]] \times \mathsf{int} \rightarrow \mathsf{ptr}[\sigma]\end{aligned}$$

The function sub now takes a *type argument* α, and then returns the appropriate specialized indexing function. For example, sub[int] returns the integer subscript function intsub.

Why is the polymorphic subscript function compiled in this way? The answer is in the fact that the various array types imply specific representations: the intarray and floatarray are unboxed, so that all of the array elements are stored contiguously in memory, while the ptrarray type assumes all elements are boxed. This improves both performance and interoperability with the slight overhead of having to provide the type argument at runtime. However, if the types are known at compile time, then even this overhead may be eliminated. For example, the source SML expression:

```
sub(x,4) + 3.14
```

[6]Informally, this type means that sub is a function that takes a pair of input arguments, an index (which is an integer) and an array whose values have type α, and returns a value of type α.

is compiled to the intermediate form

```
sub[float](x,4) + 3.14
```

since the result of the `sub` expression is constrained to be a float. This could be further inlined in the optimization phase to be

```
floatsub(x,4) + 3.14
```

Additionally, dynamic type dispatch may be used to flatten function arguments[7], flatten or align structures, etc. Furthermore, dynamic type dispatch may enable tag-free garbage collection, if unknown types are passed to the collector at runtime. All of these things enhance both speed and interoperability, since C functions expect untagged arguments in registers and may require aligned structures.

3.3.2 *Types and Intermediate Forms.* The third goal, that of propagating type information to intermediate forms, is enabled by dynamic type dispatch since intermediate forms must be able to manipulate some representation of types, as in `sub` function, above. A natural extension to this requirement is that intermediate terms themselves must be well-typed. This not only aids in the compilation of dispatch, but also improves correctness, as intermediate forms may also be type checked.

What type-system is then appropriate? I.e., how do we assign a type to a construct like `typecase`? Morrisett chooses to provide a type-system that allows type dispatch at the *type* level. For example, the function `sub` above has the type:

$$\mathsf{SpclArray}[\alpha]\times\mathsf{int}\to\alpha$$

where the specialized array constructor SpclArray is defined using the Typecase *constructor*:

$$\begin{array}{rl} \mathsf{SpclArray}[\alpha] = & \mathsf{Typecase}\ \alpha\ \mathsf{of} \\ & \ \ \mathsf{int => intarray} \\ & \mid \mathsf{float => floatarray} \\ & \mid \mathsf{ptr}[\sigma]\ \mathsf{=> ptrarray}[\mathsf{ptr}[\sigma]] \end{array}$$

The definition of the constructor parallels the term: if the parameter is instantiated with int, then the resulting type is an intarray, if the parameter is float, then the resulting type is a floatarray, etc.

This is quite different than type-systems that we've seen before. In monomorphic systems, types simply serve to name a set of values. In a polymorphic type system, types may be parameterized by type variables which essentially serve as indexes into a family of structurally-related types. Morrisett's system generalizes the polymorphic one as type variables can serve not only as indexes, but also as part of a more general computation. Despite the added complexity, Morrisett is able to prove that this type system is *sound* and that type checking is *decidable*.

3.4 The Implementation

TIL compiles SML source programs to DEC Alpha assembly language, which is then translated to standard object files by the system assembler. Compilation consists

[7]SML functions take a single argument; functions with multiple arguments are simulated by having that argument be a tuple. This implies a boxed representation; flattening unboxes the tuple into separate arguments which may be passed in registers.

of several stages; each stage translates the source language to a more restricted target language, which serves as the source language for the next stage.

The first phase borrows the ML Kit [Birkedal et al. 1993] front-end to compile from SML to a core language called Lambda. This serves to parse and assign types to the terms of the program, as well as eliminate pattern matching.

The next phase translates Lambda into a typed language called Lmli, which is based upon the concepts described above. Lmli is composed of three key constructs: terms, constructors, and kinds. Constructors serve to *name* the types of Lmli terms, while kinds serve as the types of constructors. As an example, the `sub` function as given above is essentially a Lmli term, while its type is a Lmli constructor (also shown above). Lmli provides constructs that enable dynamic type dispatch, such as `typecase`, and others that enable type-directed optimizations, such as argument flattening. Furthermore, a number of constructors exist for efficiently compiling SML `datatypes`.

The Lmli code is then translated to Lmli-Bform, which is a strict subset of Lmli structured so as to simplify optimization. A number of standard optimizations are performed, including uncurrying, dead-code elimination, hoisting, inlining, etc. These are largely the work of Tarditi [1996]. Each optimization maps Lmli-Bform into Lmli-Bform, and may be performed multiple times.

Following optimization, the code is closure converted, which serves to decouple dependencies between a function and the context of its definition. The result is a further refinement of Lmli-Bform called Lmli-Close. Because Lmli-Bform and Lmli-Close are strict subsets of Lmli, the Lmli type-checker may be applied at each step to verify the correctness of the code.

The Lmli-Bform code is then translated to an untyped version called UbForm. Variables are annotated with one of four possible runtime representations: integer, float, pointer to heap object, or unknown. Variables with unknown representation are labeled with other UBform variables that will contain the representation information at runtime. At this stage, the distinction between computation at the constructor and term level is erased.

Ubform code is translated into Rtl, or Register Transfer Language. The language is a RISC-style instruction set with a few heavyweight operations (function call and return, and interprocedural goto for exceptions) with an infinite number of registers. The final stage translates Rtl to Alpha assembly. The process includes register allocation and the generation of tables for the tag-free garbage collector. The resulting assembly is processed by the system assembler and linked into the runtime system.

3.4.1 *Performance.* Morrisett studied the performance of TIL in relation to SML/NJ, one of the most popular and well-studied SML compilers. For the benchmark programs (which are hopefully "typical"), the code compiled by TIL is typically two times faster than code compiled by SML/NJ, with comparable size. In addition, TIL programs typically require less memory, and in particular, less heap allocation. On the other hand, TIL compilation times are typically 6 times slower than SML/NJ. Morrisett also performed measurements that showed type-directed flattening in particular resulted in, on average, a 42% speedup and 50% reduction in heap allocation over non-flattened TIL code.

3.5 The Use of Types in TIL

TIL clearly meets its goals of compiling in a type-directed way to fast, interoperable code. Let us look more closely at how types play a part in achieving these goals.

Much of the novelty of TIL lies in the various forms of the typed intermediate language Lmli. One of the key features of Lmli is that it makes explicit much of the information necessary to perform optimizations through the use of types. For example, boxing and unboxing of values is based on type: a value of float type is a 64-bit IEEE-standard floating point value, while a singleton record containing a float is a boxed 64-bit floating point value. As another example, note that certain types represent "small" values and constructors which may fit in registers, while the remainder specify large ones which must be heap-allocated. Small values consist of variables, integers, floats, enums, etc., while large values are records of small values, strings, functions, and a few others. During optimization, all large values are bound to variables. This serves to maximize sharing as well as name intermediate results of computation, assisting optimization.

Another innovation of TIL is that types are preserved during much of the compilation, especially during the various refinements of Lmli. While retaining type information may limit expressibility (thus slowing down compilation time, as we have seen), it enhances reliability: each translation of the code may be type-checked for correctness. Morrisett's experience was that this greatly decreased the number of bugs in TIL. To enable this feature, Morrisett performed the painstaking task of theoretically developing λ_i^{ML}, upon which Lmli is based, to show that the type system is sound with respect to the semantics of the language, and that type-checking is decidable. Without this formal development, the correctness of the compiler would be in question.

Finally, preserving types throughout compilation enables the creation of appropriate runtime representation information for use in tag-free garbage collection. Tag-free GC is a large part of the greater interoperability and precision of TIL.

4. JAVA AND THE JVM

I shall now examine the Java language system. Java defines two parts: the proper language (called simply *Java*) and the virtual machine (called the *JVM*). Compliant systems must compile Java programs into JVM programs. These JVM programs are then executed by an interpreter.

4.1 Goals

The reason for this approach is that Java's prime purpose is to enable "mobile" programs. The intent is to provide a framework in which code can be obtained from anyone (often through a network), and safely executed on the local machine. Java is most often used to code *Applets*, which are small programs obtained and run by a web browser. A number of properties are desirable:

—**Safety**. In short, a program should not be able to subjugate the machine upon which it is executing; this mitigates the need to trust the provider of the code. Subjugation in this sense could mean the halting of the machine, the destruction of resources on the machine (e.g. disk), the communication of secure information to a third party, etc.

—**Platform Independence and Mobility**. This is especially important in the case of Applets: an Applet should run on anyone's browser, whatever the architecture.

—**General Purpose**. Java should not simply be a "toy" language but a full-featured language with good library support.

—**Easy to Learn**. With the number of full-featured languages that already exist, Java needs to be easy to learn for it to be adopted.

4.2 Language Features

Java resembles C++ in that it shares many of the object-oriented and syntactic features of that language. In fact, the Java designers had originally intended to use C++ as the language for mobile programming, but found that C++ lacked a some desirable features, such as garbage collection and exceptions, and possessed some objectionable ones, such as multiple inheritance and the general cast operator (inherited from C, see Section 2.2). Java was crafted to address these concerns.

Java programs consist of one or more `class` declarations, which serve to define a particular flavor of object. The class defines data that characterizes its objects, as well as functions, or *methods*, that may operate on those objects. A Java program begins by executing an initial function which may create objects that then interact through method calls. In addition to object types, Java provides a number of primitive types, such as `int`, `long`, etc. for efficiency.

A number of Java keywords will be used in this section that will not be explained; a more detailed, informal description of the Java language is given in [Arnold and Gosling 1996].

4.2.1 *The Type System.* Java is strongly typed, and for the most part, statically type-checked. Like C++, Java's type system makes using of subtyping, and in particular satisfies the *subsumption rule*: if τ_1 is a subtype of τ_2 (written $\tau_1 \leq \tau_2$), and a term e has type τ_1, then e can be used in a context expecting type τ_2. In general, cases of the subsumption rule can be type-checked statically. .

Nonetheless, a few runtime checks are necessary. In Java, all objects are manipulated *by reference*; an object reference is essentially a pointer to an object. Therefore, it possible for the subsumption rule to apply: if a reference `A` has type τ_1, at runtime it may point to an object with type $\leq \tau_1$, say τ_2. While the object is being referred to as of type τ_1, its τ_2 properties are masked. To reveal these properties, the programmer would like to cast the object back to its type τ_2. This kind of coercion is performed in Java with the cast (τ_2)`A`, and must be checked at runtime.

A similar runtime check results from Java's use of the covariant array subtyping rule, which states that if $\tau_1 \leq \tau_2$ then $\tau_1\ \mathsf{array} \leq \tau_2\ \mathsf{array}$. This requires that all array stores be dynamically checked, since an τ_1 array could be referred to as if it were a τ_2 array and thereby statically accept the storing of an object of type τ_2. This store is illegal since the array is actually of type τ_1, and cannot be caught until runtime.

A third check verifies that arrays are properly indexed at runtime.

4.2.2 *The JVM execution model.* As mentioned, the Java compiler translates Java programs into JVM programs; These take the form of *classfiles*; each classfile describes a single Java class. The complete classfile specification may be found in [Lindholm and Yellin 1995], chapter 4.

The three main parts of a classfile are the constant pool, the field definitions, and the method definitions. The constant pool consists of data known at compile-time, which includes program values (literals), as well as meta-information, like the names and declarations of methods and fields. This latter form of information is pointed at by the methods and fields sections, which serve to define the fields and methods of the class. In addition, both of these sections (as well as the entire classfile itself) have *attributes* associated with them. For methods, the key attribute is the *Code* attribute which contains the actual JVM instructions (called "bytecodes") that define the method.

During execution, classfiles are loaded on an as needed basis. That is, execution begins with a particular, specified class (within the method called `main`), and as the code is interpreted, other classes are loaded as they are referred to in the code. Each time a classfile is loaded, it is *verified* just before it is linked into the running system. The verification procedure makes sure that a number of static and structural constraints are satisfied thus ensuring the integrity of the file; this is described in more detail in the next section. At this point the class is linked and may be executed.

The JVM is a stack-based machine. Each method has its own operand stack, which consists of records of 32-bit values, and is bounded at compile time. There is also a runtime stack which accumulates the results of nested function calls, which cannot be bounded at compile time and may overflow at runtime. JVM instructions are often typed; for instance, the instructions *iadd*, and *ladd* specify the addition operation on (32 bit) integers, and long (64 bit) integers, respectively. This type information is not associated with the operand stack values at runtime, but verified statically.

4.2.3 *Verification.* The verification phase may be thought of as a form of static typechecking for the classfiles; it verifies that the classfile could have been generated from a valid Java compiler by verifying certain static and structural constraints. The static constraints define the *well-formedness* of the file. The structural constraints define valid relationships between JVM instructions present in the code array.

The static constraints are straightforward. For instance, all references from other sections of the classfile into the constant pool must be well typed (e.g., the constant pool record pointed at by the ConstantValue field attribute must actually specify a constant value – a `long`, `double`, `int`, etc.). All bytecodes in the code segment must define valid Java instructions, and all jump and entry points to the code array must be at the beginning of valid instructions. Types are verified for some instructions, e.g. the *atype* operand of each *newarray* instruction must be a valid primitive type (`boolean`, `char`, `int`, etc.). Constant bounds are also enforced, e.g. the *anewarray* instruction may only create an array with no more than 255 dimensions.

The structural constraints apply entirely to the bytecodes. Here is a brief (incomplete) listing:

—**Typing constraints**:
  —Each instruction may only operate on the correct number of arguments (in the

local variable section and the operand stack), and these arguments must have the proper type.
—Arguments given to a method invocation must be consistent with its definition. Similarly, return types are also checked.

—**Initialization constraints**. Initialization is a complicated procedure, as constructors and methods called from constructors may have access to partially uninitialized objects. A number of constraints are enforced in this area.

—**Access restrictions**. For example, `abstract` methods must never be invoked. Access to `protected` members is restricted to classes of the invoking class or its subtypes.

—**Bounded operand stack**. This is an interesting constraint. It requires that the size and type of the contents of the operand stack must be the same at every possible invocation of a particular instruction. It turns out that normal Java code respects this condition, but the use of the `finally` construct makes the checking somewhat more difficult. This seems like a useful feature to have as it effectively bounds the size of the operand stack for a given method invocation. Verification of this constraint is the most complicated of the process.

4.3 Comparing Implementation to Design

Let us look at how the language system design addresses the goals, noted above. First of all, Java is easy to learn in that it is based upon a widely used, popular language – C++ (which itself is based on a *more* widely used and popular language, C). This means that for many it is easy to make the transition, and for the rest, there is good reason to do so.

Secondly, Java is clearly general purpose in both language expressibility and library support. Provided libraries include GUI-support, networking primitives, and useful datastructures. In addition to standard operations, the language contains primitives for multi-threading and exceptions.

Platform-independence is achieved through the use of an interpreter for an abstract (or virtual) machine, rather than a compiler. This way, a Java application programmer needs to write only one program, and the resulting classfiles are executable on any machine that has a JVM interpreter. How is this different than simply providing a compiler for each architecture? The JVM approach also achieves mobility: a program compiled on architecture A can be run on architecture B without change.

Dynamic linking is necessary for mobility as well, for two reasons. For one, mobile programs need access to resources on the machine upon which they execute. This machine is not known at compile-time, so access to these resources, in the form of libraries, must be resolved at runtime. Secondly, a program may need Java classes not available on the executing machine. This is realistic in that most nontrivial programs consist of more than one class, yet the HTML interface to Java Applets only provides access to a single, "root", class. This class is then responsible for initiating the loading of supporting classes, which are then available to root class.

A number of runtime checks help preserve safety, in particular, array indexes are validated as in range to prevent programs from changing the contents of memory outside of legally defined array bounds. However, Java's safety relies heavily on two aspects of the Java design: strong typing in combination with garbage collection, and classfile verification.

4.4 The Use of Types in Java

Java language is strongly typed. This means that only valid object references may be obtained by a program at runtime, and not constructed from integers through encoding tricks, as in C. Furthermore, because Java is garbage collected, there is no way to obtain a "dangling-pointer;" in languages like C, such a pointer is obtained by allocating an object and then returning it to the system while retaining the pointer. At a later time, the system may reallocate the memory, and thus the retained pointer has access to inappropriate information. With garbage collection, no such activity is possible, as an object's memory is only returned to the system via the garbage collector when no pointers exist to the object. Thus, there is no means within Java to execute arbitrary machine code, since only valid memory locations may be named, thus providing a *namespace*-based protection mechanism, as opposed to an *address-space*-based one, as in many operating systems (like UNIX, unlike the MacOS and Windows)[8].

Java's use of static typing eliminates many runtime type-checks, and reduces needed runtime representation information for primitive types. However, Java's two-fold design implies that all of the safety properties guaranteed by the *Java* type system are only valid for *JVM* programs generated by a correct Java compiler. This motivates the verification phase: when the JVM interpreter loads a classfile, it must verify that the classfile has the properties of a valid Java class before it can safely execute the class code.

Much of the verification phase is enabled by the use of types in the bytecodes and in the classfile description. As mentioned, most of the bytecode instructions imply the type of their operands, and type signatures of methods and fields are present in the classfile. Most of the JVM type-checking may be done statically by the verifier before execution of the bytecodes begins, thereby eliminating performance-degrading runtime checks.

4.4.1 *Shortcomings.* Despite this effort, security holes have been found in Java. One difficulty is that the JVM bytecodes are strictly richer than the original Java source. In particular, unsafe[9] operations are expressible in the bytecodes that are not expressible in Java. While some of these unsafe code fragments are detected by the verifier, such as those with unbounded operand stacks, no formal proof exists showing that *all* of them are. Dean [1997] points out a serious error missed by the verifier in [Dean 1997].

5. ERLANG

Erlang is the product of Ericsson Telecomm, the largest telecommunications manufacturer in the world. Language research leading to the development of Erlang began in the mid-eighties with some experiments that examined the facility of various languages in programming small telephone exchanges [Dacker et al. 1986].

[8]This is has found to be untrue in practice; see Section 4.4.1.

[9]What this means is somewhat unclear, as Java does not have a well-defined security model for which to reference the term "unsafe".

These experiments led to the development of an interpreter, written in Prolog, for a language that eventually evolved into Erlang [Armstrong et al. 1992]. Since that time, the implementation of Erlang has been separated entirely from Prolog, and a number of abstract machine implementations exist [Armstrong et al. 1992; Hausman 1994]. A description of the development of Erlang, in addition to a brief tutorial, may be found in [Armstrong 1997a].

5.1 Language Features

Erlang is a functional language that shares many of the features of SML. Erlang programs consist of a series of function definitions which interoperate at runtime. In Erlang, there is no "destructive assignment"; this means that variables may only be bound to a value once. The basic datastructures consist of primitive types (like atoms[10], integers and floats), as well as lists (allowing groupings of variable length) and tuples (for groupings of fixed length). Erlang also supports a basic module system that allows only certain functions from the module to be visible outside of it, similar to, but more primitive than, SML's module system. Erlang also supports pattern matching. There are no explicit datatype declarations in Erlang, but such types are simulated using atoms within tuples, since atoms may be matched in patterns. For example, in Erlang the tuple `{human,"bob"}` is analogous to `Human("bob")` in SML.

Erlang provides language support for lightweight processes and communication between them. This facility is also used to implement a form of exception handling, as well as mutual exclusion[11].

The semantics of function invocation allow Erlang to have dynamic code updates; that is, code updates may be safely loaded into a running system. This has the effect that it's possible for an old and a new version of a function to be running at the same time! This is enabled by the fact that function calls which are qualified by the module in which they reside cause a load of the latest version of the code, while unqualified names refer to the version of code in the currently executing module.

Work is currently underway to add higher-order functions to Erlang. Combined with message passing, this makes code replacement cleaner.

A detailed description of the language and number of nontrivial sample applications may be found in [Armstrong et al. 1996].

5.1.1 *The Type System.* Erlang's type system is very similar to that of Scheme: type checks are all performed dynamically, and types are defined by the use of type predicates which may be present as side conditions (called "guards") during pattern matching. Erlang is strongly typed.

Currently, work is being done to provide a "soft" type system for Erlang [Marlow and Wadler 1997]. The type system supports subtyping and declaration-free recursive types using subtyping constraints, and is similar to one explored by Aiken

[10]These are the same as atoms in Scheme. Atoms are essentially first class names which are differentiated from variables lexically: atoms consist of all lower-case letters and variables are all upper-case

[11]It is this capacity that Erlang is not "purely" functional, as an event loop may be coded as a single function with no arguments, and the result of executing that function will be independent of its arguments.

and Wimmers [1993]. Types are inferred at compile time and compared against programmer annotations. The annotations serve as checkable "documentation" (similar to `signatures` in SML), but errors found by the checker are not enforced at runtime, hence the term "soft." The system is not currently publicly available; more information may be found at `http://solander.dcs.gla.ac.uk/erltc/`.

5.2 Goals

Erlang was developed for telecommunications applications, which demand a number of essential characteristics [Armstrong 1997b]:

—**Continuous operation**; in particular, we should be able to change code in a running system.

—**a Notion of concurrency** with fast context switching/message passing, low memory overhead per process/task, and support for thousands of processes.

—**Robust**; in particular, all objects should be dynamically sized (no fixed-length buffers that could overrun), and there should be no memory leaks/fragmentation. In addition, there should be no "global" errors (errors should remain localized, and not propagate throughout the system), and means should exist to be able to recover from SW and HW errors

—**Soft real-time**.

In addition, the designers wanted Erlang programs to be easy to develop and maintain, and to integrate easily with other languages. In the next two subsections, I will discuss how the two Erlang implementations met these goals.

5.3 the JAM

The first implementation of Erlang was written in Prolog. After the Erlang language was determined suitable by its users, attempts to improve its performance were undertaken. This led to the development of the JAM, an abstract machine based on the WAM[12] with primitives added for concurrency and exception handling. The JAM is described in [Armstrong et al. 1992].

The JAM is stack-based: all instructions retrieve arguments and place return values on a stack. Dynamic data is stored in a heap that is managed with stop and copy garbage collection. Note that because there is no destructive assignment and no means of recursive data definitions in Erlang, there cannot be any circular datastructures. In addition, heap and stack areas are dynamically sized.

The JAM employs data tagging and boxing (see Section 3 for more); for small objects (like integers), the value part is the actual object, while for larger objects, the value part points at the object as stored in the heap. Lists are represented with "cons" cells, as is typical for functional languages, while an n-tuple is represented with a header object indicating the length of the tuple immediately followed by the n (tag,value) pairs of the tuple.

Concurrency is implemented within the virtual machine directly, without direct OS support. Processes are kept on a scheduler queue and scheduled using a *fair* policy. Each process has its own stack and heap, and JAM communication primitives

[12] or Warren Abstract Machine [Warren 1983], the de-facto standard machine used to implement Prolog interpreters.

serve to copy data from the sender's data area to the receiver's. Each process has a receive queue that holds received messages; the receiver process polls the queue to determine if new messages have arrived.

JAM code is stored on per-module basis in files consisting of JAM instructions. Dynamic loading of modules is supported through the use of an *export table*. If the *call* instruction specifies a *remote* procedure as its object, then the export table is used to resolve the address of that procedure; otherwise addresses are resolved at compile time.

5.4 the BEAM

After the JAM was completed, it was still determined to be too slow, and so the BEAM (or Bogdans's Erlang Abstract Machine) was crafted to replace it [Hausman 1994]. The BEAM achieves its performance increase by replacing the use of abstract machine instructions with native code.

Erlang code is compiled in a two step process: Erlang modules are first compiled to C code, which is then compiled by `gcc` into object files. `Gcc` is used because of a number of C-extensions it supports: first-class labels, local labels, and register-allocated global variables. At then end of compilation, each module consists of a single C function with labels designating each Erlang function in the module. The code at the outset of the function serves to initialize the module. This is depicted in Figure 2.

```
void bob()
{
  /* Initialization code -- load the labels for
     exported functions into the export table */
  extern export_table* tab;
  extern void add_fun();
  add_fun(foo);
  add_fun(bar);

  /* more stuff here */
  return;

  /* Code segment */
foo:
  /* code for foo here */
bar:
  /* code for bar here */
}
```

Fig. 2. C code that results from compiling the Erlang module `bob` which consists of two functions, `foo` and `bar`

Each object file for a module is dynamically linked into the executing BEAM, and the function naming the module is called. This executes its initialization code, which among other things loads the code locations of all exported functions into the export table.

Abstract machine instructions consist of C-macros. A number of global variables are used as the registers of the machine. These are used for parameter passing and for storing return-values, as well as for temporary values. An explicit operand stack is no longer used, but a call-stack is used to track the results of nested function calls. Each Erlang process has its own call stack, heap, and message queue. Erlang processes are scheduled fairly, using the UNIX alarm facility to enforce time slices.

Memory management in the BEAM resembles that of the JAM. The data representation of objects on the heap is as (tag,value) pairs, and the heap and stack are re-sizable. Simple stop-and-copy garbage collection is used.

Performance comparisons in [Hausman 1994] show that the BEAM comes within a factor of two of an optimized hand-coded C implementation of the benchmark functions.

5.5 Comparing Goals to Implementation

Both Erlang implementations effectively meet the goals described above.

Erlang programs may receive dynamic code updates by exploiting its module system. Erlang exposes many errors to the programmer, including failed pattern matches, type errors, hardware exceptions, etc. This allows programs to recover and keep operating. Combined, these features facilitate continuous operation.

The language provides low-cost primitives for creating processes and passing messages between them, thereby enabling concurrency. Message passing translates essentially to `bcopy` within the same (Unix) process, which is a highly-optimized operation on most machines. Each Erlang process has little state, so context-switching is fast.

A number of goals are enabled by the garbage collector. Memory usage by processes is dynamically sized by the GC, minimizing resource use. GC combined with strong typing also prevents memory leaks and dangling pointers, also exploited by both Java and SML. Erlang's using of a copying garbage collector also guarantees that memory will always be used efficiently, and not fragmented. All of these characteristics combined with low-overhead message passing enables Erlang's requirement for thousands of concurrent processes.

Erlang's functional nature combined with modules and GC makes its programs easier to develop and debug (as determined by practical experience at Ericsson), even on a large scale. Furthermore, by compiling Erlang to C, as is done with the BEAM, Erlang programs may integrate with many other languages[13].

5.6 Erlang's Use of Types

Erlang's use of dynamic typing provides the benefit of greater program expressibility, without the benefit of certain performance and a priori safety guarantees made by static typing. Some of the missing safety properties are compensated for by Erlang's error handling system. Interestingly, experience with Erlang's new type system [Marlow and Wadler 1997] has shown that the majority of Erlang's base-library and kernel code may be statically type-checked with no or only slight modification [Armstrong 1997a]. However, certain functions, such as `apply`[14], may not be statically typed.

---

[13] Of course, calling other functions would require data format translation, as described in Section 3.

[14] `apply` takes as its arguments a module name `M`, a function name `F`, and a list of variably-typed elements `L`. The result of executing `apply` is the same applying function `F` from module `M` to the

It is not clear from the published material whether Erlang's implementation takes advantage of types in its intermediate forms, as do Java and TIL. JAM and BEAM instructions do assume types of their arguments, but these types are not checked until runtime, due to Erlang's use of dynamic typing. Furthermore, the BEAM inherits the type system used by its intermediate language, C, but circumvents it by, for example, using labels to define function code, rather than using standard (typeable) C function definitions. It is possible that dynamic type checks could be eliminated during an optimization phase if the types are known at compile time. Complete adoption of the new static type system might make this information available to the optimizer.

Erlang's use of strong typing combined with garbage collection serves to make Erlang programs more safe, as described for Java in Section 4.4.

6. DISCUSSION

Having looked each of the systems singly, it is interesting to see how techniques used by one system may be applied to another. For each system, I will examine whether its implementation might be improved by drawing from techniques utilized by the others.

6.1 TIL

TIL seems to form the basis of many of the techniques exploited by the other systems. It takes full advantage of strong typing combined with garbage collection (since the TIL compiler is written in SML) to prevent memory management errors. More importantly, it draws on the formal properties of types to prove the correctness of the implementation, as well as to enable optimization and tag-free GC, all to the benefit of the compiled code. However, this framework is not without overhead, resulting in fairly high compile times, thereby making it unattractive to systems where this is unacceptable (such as in JIT[15] compilers).

In general, the technique of typing intermediate representations to fortify the correctness of compilation could be exploited by any compiler. Providing such verification steps within a computation is a well-advised software-engineering principle [Maguire 1993].

TIL has much to offer the other two systems.

6.2 Java

Java's prime weakness seems to be its use of bytecodes, specifically the fact that the byte codes are richer than the Java language. Therefore, some other representation, such as abstract syntax trees (AST's) or other form of intermediate language may be appropriate. TIL can serve as an example in this regard. A well-defined intermediate-language with a proper semantics can be proven equivalent to its source language, as is the case for Lmli in TIL. Using this approach, the verifier would first check the structural integrity of the received AST (since some wire-format of the AST would have to be used to transmit the data), and

argument list `L`.

[15]JIT stands for "Just In Time." JIT compilers form part of a running program and generate code dynamically.

then would proceed to type-check it, verifying the semantic integrity. On the other hand, it can be argued that AST's are more difficult to interpret than bytecodes, thereby potentially sacrificing performance. However, better performance is not one of Sun's stated goals for Java.

The Erlang BEAM's approach of compiling to C and then to machine code might also provide an alternative to the use of bytecodes which would greatly enhance performance. To preserve code mobility, programs could be stored in so-called "fat" binaries, in which code from several architectures is bundled into a single executable[16]. This has the problem that executable size grows as new architectures are introduced and added to the binary, which is not desirable when loading code over a network. Also, this requires that compilers be able to compile for *every* supported architecture! Another approach would be to have separate repositories for programs compiled for different architectures; a web browser would then just have to be a little more savvy as to where to search for the appropriate version. This would greatly enhance the performance of Java programs, as native code can be many times faster than interpreted code. Without additional mechanism, this approach would compromise safety. However, the use of digital signatures, as in [Bershad et al. 1995], or proof-carrying code, as in [Lee and Necula 1996], might address this issue.

It has been shown [Dean 1997; Dean et al. 1996] that dynamic linking in the presence of static typing can compromise security. This calls into conflict two of the Java goals: mobility and safety. Again, TIL may serve as an example for resolving this issue in its use of formal methods: given a formal semantics for Java and the JVM, we would like to be able to prove that the design and implementation of dynamic linking are in fact safe relative to some well-defined security model. Currently no such formal semantics exist, but Dean [1997] has begun the process of developing formalisms for both simple JVM code and dynamic linking.

Presuming that byte codes remain the wire format of choice, a complete JVM semantics is desirable. One of the difficulties in the task of providing such a semantics is the complexity of the Java bytecodes: there are literally hundreds of bytecodes, some with quite subtle meanings. Erlang's JAM may provide guidance here. There are only a small number of instructions, and these instructions encode a model of computation that is fairly well studied (that is, a functional style based on the lambda-calculus) and developed (the JAM is derived from the WAM [Warren 1983]). On the other hand, part of the reason for the JAM's small instruction set is the lack of type information coded into the instructions; each instruction performs dynamic type-checking. However, it is not clear whether this would be a hindrance to performance. For one, the JVM already keeps almost complete runtime type information to facilitate garbage collection and its already existent dynamic checks. Secondly, it is not clear whether runtime type-checks make up a significant portion of computation time; it would be interesting to measure this. Furthermore, the static checking provided by the Java language would still guarantee safety for programs developed from a Java compiler, and the runtime checks would make much of the verification phase unnecessary.

[16] This is Apple's approach while it is transitioning Macinstosh systems that use Motorola 68K chips to the Power PC.

6.3 Erlang

In general, Erlang's requirements are quite different from those of the other two systems. However, some things could still be done to improve its implementation.

It is not clear what Erlang's security requirements are. Both abstract machines allow dynamic linking, but nothing I read indicated that loaded code is verified before it is executed. Therefore, the loading process must somehow be secure (such as only through physical access to the machine's disk). Should this situation change, a verification step as performed by Java would be helpful for Erlang.

The use of dynamic typing makes a number of the statically-enforced guarantees made by Java and TIL impossible in Erlang. However, as stated, a static type system is being developed for Erlang which correctly checks most core Erlang code. If adopted, performance and security of the resulting code may be improved. Within the implementation, type information garnered at compile-time might facilitate certain optimizations, such as the elimination of dynamic type checks, as well as some of the other optimizations described by Morrisett. Erlang's BEAM might be able to then make more strict use of the type system present in its intermediate language, C. However, as it stands, some Erlang programs cannot be properly type-checked, so the question of the importance of those programs arises. I can only speculate from what I've read, but it appears that the Erlang team is not willing to change any legacy code.

7. CONCLUSIONS

Types may be used effectively by language system designers to strengthen their implementations. Greg Morrisett illustrates with TIL that a formal approach to compiler design that relies on the use of typed intermediate representations can greatly enhance the correctness, speed, and interoperability of the resulting code. Java uses two levels of static checking (once by the compiler and once by the verifier) to guarantee the safety of mobile programs, for both the program and the machine that it executes on. While Erlang currently does not make use of types in its intermediate representations, work with a new static type system could make this possible, enabling benefits realized by the other two systems. Of the three, TIL makes the most extensive use of types and formal methods in its implementation, and as a result has the greatest basis for correctness. A lack of a formal semantics for Java and its security policy makes its implementation more suspect, and in fact a number of errors have already been uncovered [Dean et al. 1996]; the need for a formal semantics seems clear [Dean 1997]. While a number of implementation decisions for Erlang are not present in the material I read (such as how code is dynamically linked), it seems that these decisions could only be strengthened by the use of types and/or formal methods.

REFERENCES

Aiken, A. and Wimmers, E. L. 1993. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*. 31–41.

Alexander, D. S., Shaw, M., Nettles, S. M., and Smith, J. M. 1997. Active Bridging. In *Conference for the Special Interest Group on Data Communications (SIGCOMM)*. ACM.

Appel, A. W. and MacQueen, D. B. 1987. A Standard ML Compiler. In *Functional Programming Languages and Computer Architecture*. Springer-Verlag, 301–324. Volume 274 of *Lecture Notes in Computer Science*.

ARMSTRONG, J. 1997a. The development of Erlang. In *SIGPLAN International Conference on Functional Programming*. ACM, 196–203.

ARMSTRONG, J. 1997b. The development of Erlang. Slides for talk at ICFP, Amsterdam.

ARMSTRONG, J., VIRDING, R., WIKSTROM, C., AND WILLIAMS, M. 1996. *Cconcurrent Programming in Erlang*, Second ed. Prentice Hall.

ARMSTRONG, J. L., DACKER, B. O., VIRDING, S. R., AND WILLIAMS, M. C. 1992. Implementing a Functional Language for Highly Parallel Real Time Applications. In *Software Engineering for Telecommuncation Systems and Services*.

ARMSTRONG, J. L., VIRDING, S. R., AND WILLIAMS, M. C. 1992. Use of Prolog for developing a new programming language. In *The Practical Application of Prolog*.

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. Addison-Wesley.

BERRY, D. 1991. The edinburgh sml library. Tech. Rep. ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University. April.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *15th Symposium of Operating Systems Principles*. ACM.

BIRKEDAL, L., ROTHWELL, N., TOFTE, M., AND TURNER, D. N. 1993. The ML kit (version 1). Tech. Rep. DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen.

CLINGER, W. AND REES, J. 1991. The Revised(4) Report on the Algorithmic Language Scheme. *Lisp Pointers 4* 3, 1–51.

DACKER, B. O., ELSHIEWY, N., HEDELAND, P., WELLIN, C. W., AND WILLIAMS, M. C. 1986. Experiments with Programming Languages and Techniques for Telecommuncations Applications. In *Software Engineering for Telecommuncation Systems and Services*.

DEAN, D. 1997. The Security of Static Typing with Dynamic Linking. In *Fourth Conference on Computer and Communications Security*. ACM.

DEAN, D., FELTEN, E. W., AND WALLACH, D. S. 1996. Java Security: From Hotjava to Netscape and Beyond. In *Symposium on Security and Privacy*. IEEE.

GUNTER, C. 1992. *Semantics of Programming Languages, Structures and Techniques*. MIT Press.

HARPER, R. AND LEE, P. 1994. Advanced languages for systems software: The Fox project in 1994. Tech. Rep. CMU-CS-94-104, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. January. (Also published as Fox Memorandum CMU-CS-FOX-94-01).

HAUSMAN, B. 1994. Turbo Erlang: Approaching the Speed of C. In *Implementations of Logic Programming Systems*, Evan Tick and Giancarlo Succi, Eds. Kluwer Academic Publishers, 119–135.

LEE, P. AND NECULA, G. 1996. Safe Kernel Extensions Without Run-Time Checking. In *Symposium on Operating Systems Design and Implementation*. ACM.

LEROY, X. 1995. *The Caml Special Light System (Release 1.10)*. INRIA, France.

LINDHOLM, T. AND YELLIN, F. 1995. *The Java Virtual Machine Specification*. Addison-Wesley.

MAGUIRE, S. 1993. *Writing Solid Code*. Microsoft-Press.

MARLOW, S. AND WADLER, P. 1997. A Practical Subtyping System for Erlang. In *SIGPLAN International Conference on Functional Programming*. ACM, 136–149.

MORRISETT, G. 1995. Compiling With Types. Tech. Rep. CMU-CS-95-226, School of Computer Science, Carnegie Mellon University. Dec.

MYERS, C., CLACK, C., AND POON, E. 1994. *Programming with Standard ML*. Prentice-Hall.

TARDITI, D. 1996. Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML. Tech. Rep. CMU-CS-97-108, School of Computer Science, Carnegie Mellon University. Dec.

WARREN, D. H. D. 1983. An abstract Prolog instruction set. Tech. Rep. 309, SRI International, Menlo Park, CA. October.