Compiling Standard ML to Java Bytecodes

Nick Benton, Andrew Kennedy, George Russell

In *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*, September 1998, Baltimore.

The following copyright notice is required by the ACM (see `http://www.acm.org/pubs/copyright_policy.html`).

Copyright © 1998 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or `permissions@acm.org`.

Compiling Standard ML to Java Bytecodes

Nick Benton Andrew Kennedy George Russell

Persimmon IT, Inc.
Cambridge, U.K.
{nick,andrew,george}@persimmon.co.uk

Abstract

MLJ compiles SML'97 into verifier-compliant Java bytecodes. Its features include type-checked interlanguage working extensions which allow ML and Java code to call each other, automatic recompilation management, compact compiled code and runtime performance which, using a 'just in time' compiling Java virtual machine, usually exceeds that of existing specialised bytecode interpreters for ML. Notable features of the compiler itself include whole-program optimisation based on rewriting, compilation of polymorphism by specialisation, a novel monadic intermediate language which expresses effect information in the type system and some interesting data representation choices.

1 Introduction

The success of Sun Microsystem's Java language [3] means that virtual machines executing Java's secure, multi-threaded, garbage-collected bytecode and supported by a capable collection of standard library classes, are now not just available for a wide range of architectures and operating systems, but are actually installed on most modern machines. The idea of compiling a functional language such as ML into Java bytecodes is thus very appealing: as well as the obvious attraction of being able to run the same compiled code on any machine with a JVM, the potential benefits of interlanguage working between Java and ML are considerable.

Many existing compilers for functional languages have the ability to call external functions written in another language (usually C). Unfortunately, differences in memory models and type systems make most of these foreign function interfaces awkward to use, limited in functionality and even type-unsafe. Consequently, although there are, for example, good functional graphics libraries which call X11, the typical functional programmer probably doesn't bother to use a C language interface to call 'everyday' library functions to, say, calculate an MD5 checksum, manipulate a GIF file or access a database. She thus either does more work than should be necessary, or gives up and uses another language. This is surely a major factor holding back the wider adoption of functional languages in 'real world' applications, and the situation is getting worse as more software has to operate in a complex environment, interacting with components written in a variety of languages, possibly wrapped up in a distributed component architecture such as CORBA, DCOM or JavaBeans.

Because the 'semantic gap' between Java and ML is smaller than that between C and ML, and because Java uses a simple scheme for dynamic linking, MLJ is able to make interlanguage working safe and straightforward. MLJ code can not only call external Java methods, but can also manipulate Java objects and declare Java classes with methods implemented in ML, which can be called from Java. Thus the MLJ programmer can not only write applets in ML, but also has instant access to standard libraries for 2 and 3D graphics, GUIs, database access, sockets and networking, concurrency, CORBA connectivity, security, servlets, sound and so on, as well as to a large and rapidly-growing collection of third-party code.

The interesting question is whether ML can be compiled into Java bytecodes which are efficient enough to be useful. Java itself is often criticised for being too slow, especially running code which does significant amounts of allocation, and its bytecodes were certainly not designed with compilation of other languages in mind. There is little opportunity for low-level backend trickery since we have no control over the garbage collector, heap layout, etc., and the requirement that compiled classes pass the Java verifier places strict type constraints on the code we generate. Furthermore, current Java virtual machines not only store activation records on a fixed-size stack, but also fail to optimise tail calls. Thus the initial prospects for generating acceptably efficient Java bytecodes from a functional language did not look good (our first very simple-minded lambda-calculus to Java translator plus an early JVM ran the nfib benchmark 40 times slower than Moscow ML), and it was clear that a practical ML to Java bytecode compiler would have to do fairly extensive optimisations. MLJ is still a work-in-progress, and there is scope for significant improvement in both compilation speed and the generated code (in particular, the current version still only optimises simple tail calls), but it is already quite usable on source programs of several thousand lines and produces code which, with a good modern JVM, usually outperforms the popular Moscow ML bytecode interpreter.

To appear in the 3rd ACM SIGPLAN Conference on Functional Programming, September 1998, Baltimore

2 Overview

2.1 Compiler phases

MLJ is intended for writing compact, self-contained applications, applets and software components and does not have the usual SML interactive read-eval-print top level. Instead, it operates much more like a traditional batch compiler. The structures which make up a project are separately parsed, typechecked, translated into our typed Monadic Intermediate Language (MIL, see section 4) and simplified. These separately compiled MIL terms are then linked together into one large MIL term which is extensively transformed before being translated into our low-level Basic Block Code (BBC, see section 7). Finally, the backend turns the BBC into a collection of compiled Java class files, which by default are placed in a single `zip` archive.

This whole-program approach to compilation is unusual, though not unique [26, 22]. It increases recompilation times considerably, but does allow us easily to produce much faster and (just as importantly for applets) smaller code. We monomorphise the whole program and can perform transformations such as inlining, dead-code elimination, arity-raising and known-function call optimisation with no regard for module boundaries, so there is no runtime cost associated with use of the module system.[1]

Appendix A contains an example of JVM code generated by MLJ.

2.2 Compilation environment

MLJ can be run entirely in batch mode or as an interactive recompilation environment. Top-level structures and signatures are stored one-per-file, as in Moscow ML (MLJ 0.1 doesn't implement functors). All compilation is driven by the need to produce one or more named Java class files: for an application this is usually just the class containing the `main` method, whilst for an applet it is usually a subclass of `java.awt.Applet`. Once these root classes and their exported names have been specified, the `make` command compiles, links and optimises the required structures using an automatic dependency analysis. There is a smart recompilation manager, similar to SML/NJ's CM [7], which ensures that only the necessary structures are recompiled when a file is changed, though the post-link optimisation phase is always performed on the whole program. Typically, the recompile time (relinking all the translated MIL structures from memory, optimising and generating code) is around two thirds of the total initial compile time (which also includes parsing, typechecking and translation into MIL).

During compilation, the compiler not only typechecks the ML code, but if any external Java classes are mentioned, their compiled representations are read (typically from within the standard `classes.zip` file) to typecheck and resolve the references.

3 The Language

MLJ compiles the functor-free subset of SML'97 [14] (including substructures, the new `where` construct, etc.), plus our non-standard extensions for interlanguage working with Java.[2] A large subset of the new standard Basis Library has been implemented.

The interlanguage features bring Java types and values into ML, whilst enforcing a separation between the two (though a Java type and an ML type may well end up the same in the compiled code). External Java types may be referred to in ML simply by using their Java names in quotation marks. Thus

```
"java.awt.Graphics" * int
```

is the type of pairs whose first component is a Java object representing a graphics context and whose second component is an ML integer. An important subtlety is that, whilst in Java all pointer (reference) types implicitly include the special value `null`, we chose to make their quoted Java type names in ML refer only to the non-`null` values. Where a value of Java reference type t may be passed from Java to ML code (as the result of an external field access or method call, or as a parameter to a method implemented in ML), then that value is given the ML type "t" `option` with the value `NONE` corresponding to Java `null` and values of the form `SOME(v)` corresponding to non-`null` values. Similarly we guarantee that any ML value of type "t" `option` will be represented by the an element of the underlying Java class. This complication allows the type system to catch statically what would otherwise be dynamic `NullPointerException`s and also gives the compiler more freedom in choosing and sharing data representations (see Section 6).

The builtin structure `Java` includes ML synonyms for common Java types and coercions between many equivalent ML and Java types, for example `Java.toString`, which converts a Java `String` into an ML `string`. With one exception (`fromWord8`), these all generate no actual bytecodes, and are included just to separate the two type systems securely in the source.

MLJ code can perform basic Java operations such as field access, method invocation and object creation by using a collection of new keywords, all of which start with an underscore '`_`' (which fits well with the existing lexical structure of SML), and we also use quotation marks for Java field and method names. Thus, for example

```
let type colour = "java.awt.Color"
    val grey = valOf (_getfield colour "gray")
in Java.toInt (_invoke "getRed" (grey))
end
```

makes `colour` be an ML synonym for the Java class `Color` and then gets the static field called `gray` from that class. MLJ reads the compiled Java class file which states that the field holds an *instance* of the class `Color` and so infers the type "`java.awt.Color` `option`" for the `_getfield` construct, because it's an external value which might be `null`. We then invoke the virtual method `getRed()` on the returned colour value to get the Java `int` value of its red component, which we convert into an ML `int`. The `valOf` is required to remove the `option` from the type of the returned value (it raises an exception if its argument is `NONE`) because MLJ does not allow method invocation on possibly-`null` Java values.

The new constructs such as `_getfield` and `_invoke` have essentially the same static semantics as their equivalents in

[1] 'Whole program compiler' sounds a bit naive, so we prefer to think of it as a 'post-link optimiser', which has a much more sophisticated ring :-)

[2] Actually, version 0.1 departs from the Definition in four places. The only significant one is that arithmetic operations do not raise the `Overflow` exception.

```
datatype Behave = B of unit -> Behave

_classtype MLButton _extends "java.awt.Button" {
 _private _field "behaviour" : Behave

 _public _method "action" (e:"java.awt.Event" option, o : Java.Object option) : Java.boolean =
  let val B(b) = _getfield "behaviour" _this
  in (_putfield "behaviour" (_this, b ()); Java.fromBool true)
  end

 _constructor (name : string, b:Behave)
              {_super(Java.fromString name); "behaviour" = b}
}
```

Figure 1: Generating a Java class from MLJ

the Java language. At a virtual method invocation, for example, the compiler searches up the class hierarchy for a method matching the given name and argument types, applying the same rules for finding the 'most specific method' by possible coercions on the arguments as Java does. (ML polymorphism isn't allowed to confuse things further as all uses of these new constructs must be implicitly or explicitly monomorphic.) This makes it easy to convert existing Java programs or code fragments into MLJ but is not intended to make MLJ a fully-fledged object-oriented extension of SML; in particular, inheritance does not induce any subtyping relation on ML types.

MLJ structures can also declare new Java classes with fields and methods having a mixture of ML and Java types and methods implemented in ML. In fact, all programs must contain at least one such class for the Java runtime system to call into; otherwise no ML code could ever get called. The extensions for declaring classes can express most of what can be expressed in the Java language, including all the access modifiers (`public`, `private`, etc.), though there are some natural restrictions, such as that static fields of ML type (or non-`option` Java reference type) must have initialisers, since there are no default values of those types, and overloading on ML types is forbidden (as two ML types may end up represented as the same Java type). Class definitions in structures are all anonymous (there are ML type names bound to them, but no Java class names) so a class that is referenced purely by ML code will end up with an internal name in the compiled program, and so not be directly accessible from external Java code.[3] However, as we've already mentioned, at least one top-level class has to be exported with a given Java name. Since exported classes can be accessed from Java, there are some not entirely trivial further restrictions concerning the types and access modifiers of their fields and methods which are necessary to ensure that anything which is visible, either directly or by inheritance, to the external Java world is of Java primitive or `option`ed Java class type.

Just to give the flavour of how Java classes may be generated from MLJ, Figure 1 shows the definition of an ML button class, similar to one which might form part of a functional GUI toolkit. `MLButton` subclasses the standard Java `Button` class and has an instance variable which is of a higher-order ML datatype `Behave`. When the button is pressed, its `action` method is called, which causes its behaviour function to be called (presumably with some side-effects), returning a new behaviour which is stored in the instance variable ready for the next click. The constructor (for which our syntax is particularly baroque) is called with an ML `string` and the initial behaviour. It starts by calling the superclass initialiser with a Java `String` and then initialises the instance variable with the supplied behaviour.

[3] Unless it uses Java's introspection capabilities, which we consider to be cheating...

4 MIL

The Monadic Intermediate Language, MIL, is the heart of the compiler. MIL is a typed language, inspired by Moggi's computational lambda calculus [15], which makes a distinction between computations and values in its type system. It has impredicative System-F-style polymorphism and refines the computation type constructor to include effect information. MIL also includes some slightly lower level features so that most of the optimisations and representation choices we wish to make can be expressed as MIL-to-MIL transformations. These include not-quite-first-class sequence types (used for multiple argument/multiple result functions and flat datatype constructors), Java types (used not just for interlanguage working, but also to express representations for ML types) and three kinds of function: local, global and closure.

Typed intermediate languages are now widely accepted to be A Good Thing [19, 22, 11]. Types aid analysis and optimisation, provide a more secure basis for correct transformations, and are required in type-directed transformations such as compilation of polymorphic equality. They also help catch compiler bugs! In our case it seems especially natural, as the Java bytecode which we eventually produce is itself typed.

The use of computational types in the intermediate language is more unusual, though similar systems have recently been proposed in [23, 11, 24] and the use of a monadic intermediate language to express strictness-based optimisations was proposed in [4]. The separation of computations and values gives MIL a pleasant equational theory (full β and η rules plus commuting conversions), which makes correct rewriting simpler. Order of evaluation is made explicit (much as it is in CPS-based intermediate representations [1]), and our refinement of computation types into different monads gives a unified framework for effect analysis and associated transformations.

V	::=	x, y, f	value variables
	|	c	constant of base type
	|	$(V_1, \ldots, V_n)$	tuple
	|	$\mathsf{in}_i \vec{V}$	injection into sum
	|	$\mathsf{in}_\chi \vec{V}$	injection into exception
	|	$\Lambda \vec{t}.V$	type abstraction
	|	$V_{\vec{\tau}}$	type application
	|	$\mathsf{fold}_\tau\, V$	recursive type introduction
	|	$\mathsf{unfold}\, V$	recursive type elimination
	|	$\pi_i V$	projection
M	::=	$\mathsf{val}\, \vec{V}$	trivial computation
	|	$\mathsf{let}\ \vec{x} \Leftarrow M_1\ \mathsf{in}\ M_2$	evaluation
	|	$V\, \vec{V}$	function application
	|	$\mathsf{ref}\, V$	reference creation
	|	$!V$	dereferencing
	|	$V_1 := V_2$	assignment
	|	$\mathsf{raise}\, V$	throw an exception
	|	$\mathsf{try}\ \vec{x} \Leftarrow M_1\ \mathsf{handle}\ y \Rightarrow M_3\ \mathsf{in}\ M_2$	evaluate and catch
	|	$\mathsf{case}\ V\ \mathsf{of}\ \mathsf{in}_1 \vec{x}_1 \Rightarrow M_1; \ldots; \mathsf{in}_n \vec{x}_n \Rightarrow M_n$	case on sum
	|	$\mathsf{case}\ V\ \mathsf{of}\ \mathsf{in}_{\chi_1} \vec{x}_1 \Rightarrow M_1; \ldots; \mathsf{in}_{\chi_n} \vec{x}_n \Rightarrow M_n; _ \Rightarrow M$	case on exception
	|	$\mathsf{case}\ V\ \mathsf{of}\ c_1 \Rightarrow M_1; \ldots; c_n \Rightarrow M_n; _ \Rightarrow M$	case on base type
	|	$\mathsf{let}\ \Lambda \vec{t}\, f_1(\vec{x}_1 : \vec{\tau}_1) = M_1 : \gamma_1; \ldots; f_n(\vec{x}_n : \vec{\tau}_n) = M_n : \gamma_n\ \mathsf{in}\ M$	(recursive) function declaration

Figure 3: Terms in MIL

τ	::=	$\mathsf{int} \mid \mathsf{char} \mid \ldots$	base types
	|	$\tau_1 \times \cdots \times \tau_n$	product
	|	$\vec{\tau}_1 + \cdots + \vec{\tau}_n$	sum
	|	$\vec{\tau} \to \gamma$	function type
	|	$\tau\ \mathsf{ref}$	reference type
	|	exn	exception
	|	$\forall \vec{t}.\tau$	polymorphic type
	|	$\mu t.\tau$	recursive type
$\vec{\tau}$	::=	$\langle \tau_1, \ldots, \tau_n \rangle$	vector of types
γ	::=	$\mathbf{T}_\varepsilon(\vec{\tau})$	computation type
ε	$\subseteq$	$\{\bot, \mathsf{throws}, \mathsf{reads}, \mathsf{writes}, \mathsf{allocs}\}$	effects

Figure 2: Types in MIL

4.1 Types and terms

Types in MIL are divided into *value* types (ranged over by τ) and *computation* types (ranged over by γ) as shown in Figure 2.[4] Value types include base types (int, char, *etc.*), products, function types with multiple arguments and results, sum types with multiple argument summands, reference types, polymorphic types and recursive types. Computation types have the form $\mathbf{T}_\varepsilon(\vec{\tau})$, indicating a computation with result types $\vec{\tau}$ and effect ε. The subset relation on effects induces a subtyping relation on types in an unsurprising way. Note that because we are only interested in compiling a call-by-value language, we have restricted function types to be from values to computations.

Terms are also divided into values (ranged over by V) and computations (ranged over by M) as shown in Figure 3. All evaluation happens in let or try. The 'Moggi let' construct

$$\mathsf{let}\ \vec{x} \Leftarrow M_1\ \mathsf{in}\ M_2$$

evaluates the computation term M_1 and binds the result to x in the scope of the computation term M_2. The construct

$$\mathsf{try}\ \vec{x} \Leftarrow M_1\ \mathsf{handle}\ y \Rightarrow M_3\ \mathsf{in}\ M_2$$

generalises this by providing a handler M_3 in which the exception raised by M_1 is bound to a variable y and the continuation M_3 is not evaluated. (It is interesting to note that this construct cannot be defined using let and the more conventional $M_1\ \mathsf{handle}\ y \Rightarrow M_2$ without recourse to a value of sum type). The construct $\mathsf{val}\, \vec{V}$ allows a value (or multiple values) to be treated as a trivial computation.

4.2 Translation from SML

The initial translation from typed SML to MIL is essentially Moggi's call-by-value translation [15]. For example, a source application $exp_1\ exp_2$ translates to

$$\mathsf{let}\ x \Leftarrow M_1\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow M_2\ \mathsf{in}\ x\ y$$

where M_1 and M_2 are the translations of exp_1 and exp_2. The translation also expands out certain features of the source language that are not present in the target: patterns are compiled into flat case constructs, records are translated as ordinary tuples (with fields sorted by label) and even the `structure` constructs of the SML module language are compiled into ordinary tuples of values. The latter relies on the impredicative polymorphism in MIL. An alternative (see [20]) would be to stratify the intermediate language in a similar way to the stratification of SML into code and module levels.

[4] Some simplifications have been made for this presentation. In particular the implementation has *mutual* recursion over multiple types; also, lower-level features that capture Java representations are omitted.

Uses of polymorphic equality are also compiled away, using a simple dictionary-passing style. One might expect that equality would be an ideal candidate for exploiting Java's virtual methods, overriding the `equals` method in classes representing ML types. But that would prevent us sharing representations between ML types with different equality functions, so doesn't seem worth doing.

5 Transformations

Most of the compiler's time is spent applying a set of MIL-to-MIL transformations to the term representing the whole program. Some of these are considered to be general-purpose 'simplification' steps, and are applied at several stages of the compilation, whilst others perform specific transformations, such as arity-raising functions.

Most of the transformations are 'obviously' meaning-preserving rewrites[5], but the tricky part is deciding when they should be applied. At the moment, most of these decisions are taken on the basis of some simple rules, rather than as the result of any sophisticated analysis. Some of these rules are type-directed whereas others involve simple properties of terms, such as the size of a subterm or number of occurrences of a variable. The effect inference is currently rather naive, particularly with regard to recursive functions and datatypes, so there are a very small number of places in the basis where we have annotated computations as pure so that they may be dead-coded in programs in which they are not referenced.

5.1 Simplification

The most basic of the transformations are essentially just the 'pure' β and η reductions and the commuting conversions one obtains from the proof theory of the computational lambda calculus [5], adapted so that large or allocating terms are not duplicated (see Figure 4). The reductions are genuine simplifications whilst the commuting conversions are reorganisations which tend to 'expose' further reductions. MLJ applies the commuting conversions exhaustively to obtain a CC-normal form from which code generation is particularly straightforward. Doing this sort of heavy rewriting on a large term can be expensive, particularly when it is done functionally as the heap turnover is then very high. Our current simplifier uses a quasi-one-pass algorithm similar to that described by Appel and Jim in [2]: it maintains an environment of variable bindings, a census to count variable occurrences and a stack of 'evaluation contexts' to perform commuting conversions efficiently. This algorithm is several times faster than our first version, but simplication still ends up being the most expensive phase because it is repeated at several stages during compilation – the total time spent in the simplifier is typically around half the recompile time.

The validity of certain rewrites depends on effect information in the types. Some of these are shown in Figure 5.

5.2 Polymorphism

Most implementations of SML compile parametric polymorphism by *boxing*, that is, by ensuring that values of type t

[5] Not that we're claiming to have justified them formally with respect to a semantics for the full language!

case-beta :
$\mathsf{case}\ \mathsf{in}_i\vec{V}\ \mathsf{of}\ \mathsf{in}_1\vec{x}_1 \Rightarrow M_1;\ \cdots;\ \mathsf{in}_n\vec{x}_n \Rightarrow M_n$
$\rightsquigarrow\ \mathsf{let}\ \vec{x}_i \Leftarrow \mathsf{val}\ \vec{V}\ \mathsf{in}\ M_i$

let-eta :
$\mathsf{let}\ \vec{x} \Leftarrow M\ \mathsf{in}\ \mathsf{val}\ \vec{x}$
$\rightsquigarrow\ M$

let-let-cc :
$\mathsf{let}\ \vec{x}_1 \Leftarrow (\mathsf{let}\ \vec{x}_2 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3$
$\rightsquigarrow\ \mathsf{let}\ \vec{x}_2 \Leftarrow M_1\ \mathsf{in}\ \mathsf{let}\ \vec{x}_2 \Leftarrow M_2\ \mathsf{in}\ M_3$

let-case-cc :
$\mathsf{let}\ \vec{y} \Leftarrow \mathsf{case}\ V\ \mathsf{of}\ \mathsf{in}_1\vec{x}_1 \Rightarrow M_1\ ;\ \cdots\ ;\ \mathsf{in}_n\vec{x}_n \Rightarrow M_n\ \ \mathsf{in}\ M$
$\rightsquigarrow\ \mathsf{let}\ f(\vec{y}) = M\ \mathsf{in}$
$\quad \mathsf{case}\ V\ \mathsf{of}$
$\quad\quad \mathsf{in}_1\vec{x}_1 \Rightarrow \mathsf{let}\ \vec{y} \Leftarrow M_1\ \mathsf{in}\ f\ \vec{y}\ ;\ \cdots\ ;$
$\quad\quad \mathsf{in}_n\vec{x}_n \Rightarrow \mathsf{let}\ \vec{y} \Leftarrow M_n\ \mathsf{in}\ f\ \vec{y}$

Figure 4: Some proof-theoretic rewrites

dead-let :
$\mathsf{let}\ \vec{x} \Leftarrow M_1\ \mathsf{in}\ M_2$
$\rightsquigarrow\ M_2$
if $\vec{x}$ not free in M_2 and $M_1 : \mathbf{T}_\varepsilon(\vec{\tau})$ for $\varepsilon \subseteq \{\mathsf{allocs}, \mathsf{reads}\}$

dead-try :
$\mathsf{try}\ \vec{x} \Leftarrow M_1\ \mathsf{handle}\ y \Rightarrow M_2\ \mathsf{in}\ M_3$
$\rightsquigarrow\ \mathsf{let}\ \vec{x} \Leftarrow M_1\ \mathsf{in}\ M_3$
if $M_1 : \mathbf{T}_\varepsilon(\vec{\tau})$ where $\mathsf{throws} \notin \varepsilon$

hoist-let :
$\mathsf{let}\ \vec{y} \Leftarrow M\ \mathsf{in}\ \mathsf{case}\ V\ \mathsf{of}\ \mathsf{in}_1\vec{x}_1 \Rightarrow M_1;\ \cdots;\ \mathsf{in}_n\vec{x}_n \Rightarrow M_n$
$\rightsquigarrow\ \mathsf{case}\ V\ \mathsf{of}$
$\quad \mathsf{in}_1\vec{x}_1 \Rightarrow M_1\ ;\ \cdots\ ;$
$\quad \mathsf{in}_i\vec{x}_i \Rightarrow \mathsf{let}\ \vec{y} \Leftarrow M\ \mathsf{in}\ M_i\ ;\ \cdots\ ;$
$\quad \mathsf{in}_n\vec{x}_n \Rightarrow M_n$
if $\vec{y}$ free only in M_i and $M : \mathbf{T}_\varepsilon(\vec{\tau})$ for $\varepsilon \subseteq \{\mathsf{allocs}, \mathsf{reads}\}$

Figure 5: Some rewrites dependent on effect information

inside a value of type $\forall t.\tau$ are represented uniformly by a pointer (they are 'boxed'). In Java, the natural way to box objects is by a free cast to `Object`, and the natural way to box primitive types is to create a heap-allocated wrapper object and cast that to `Object`. Unboxing then involves using Java's `checkcast` bytecode to cast back down again and in the case of primitive types, extracting a field. Done naively, this kind of boxing can be extremely inefficient and there are a number of papers which address the question of how to place the coercions to reduce the cost of boxing (*e.g.*[13, 9]). For an early version of our compiler we implemented a recent and moderately sophisticated graph-based algorithm for coercion placement [12].

Whilst the graph-based algorithm worked fairly well, we have more recently taken a more radical and straightforward approach: removing polymorphism entirely. This is possible firstly because we have the whole program to work with, and secondly because it is a property of Standard ML that the finite number of types at which a polymorphic function is used can be determined statically. In languages with polymorphic recursion (such as recent versions of Haskell) this property does not hold: the types at which a function is used may not be known until run-time.

Each polymorphic function is specialised to produce a separate version for each type instance at which it is used. In the worst case, this can produce an exponential blowup in code size, but our experience is that this does not actually happen in practice. Three reasons can be cited for this. First, we specialise not with respect to source types but with respect to the final Java representations, for which much sharing of types occurs. For example, suppose that the `filter` function is used on lists containing elements of two different datatypes, both of which are represented by the 'universal sum' class discussed in Section 6. Then only one version of `filter` is required.[6] Second, boxing and unboxing coercions introduce a certain amount of code blowup of their own and this is avoided. Third, polymorphic functions tend to be small, so the cost of duplicating them is often not great. Indeed, not only are many polymorphic functions inlined away prior to monomorphising, but *after* monomorphising it is often the case that a particular instance of a function is used only once and is consequently inlined and subject to further simplification.

5.3 Arity raising

Heap allocation is expensive. We try to avoid creating tuples and closures by replacing curried functions and functions accepting tuples with functions taking multiple arguments, and to flatten sums-of-products datatypes by using multiple argument constructors. We also remove the type unit (nullary product) entirely. These transformations should ideally be driven by information about how values are used, but we find that fairly simple-minded application of type isomorphisms such as the following

$$\begin{array}{lcl}
\tau_1 \times \tau_2 \to \gamma & \cong & \langle \tau_1, \tau_2 \rangle \to \gamma \\
\tau_1 \to \mathbf{T}_\emptyset(\tau_2 \to \gamma) & \cong & \langle \tau_1, \tau_2 \rangle \to \gamma \\
\tau_1 \times \tau_2 + \tau_3 \times \tau_4 & \cong & \langle \tau_1, \tau_2 \rangle + \langle \tau_3, \tau_4 \rangle \\
\tau \times \mathsf{unit} & \cong & \tau \\
\mathsf{unit} \to \gamma & \cong & \langle\rangle \to \gamma \\
\mathbf{T}_\varepsilon\langle \mathsf{unit} \rangle & \cong & \mathbf{T}_\varepsilon\langle\rangle
\end{array}$$

[6] This can itself be regarded as a kind of boxing, since we're using uniform (shared) representations for certain ML types, but note that we never box primitive types.

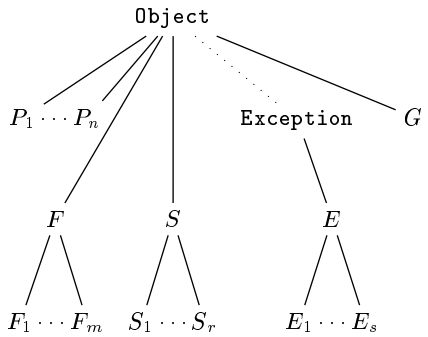

Figure 6: Java classes representing MIL types

combined with the other rewrites, produces a significant improvement in most programs. For example, the Boyer-Moore benchmark runs in 4 seconds with all optimisation enabled but takes 6 seconds when tuple-argument arity raising is turned off. Worse, it crashes with a stack overflow when de-currying is also disabled because MLJ is then unable to use a `goto` instruction in place of a tail call.

Observe that de-currying depends upon effect information in the types (including termination), for otherwise it would be unsound.

6 Data representation

6.1 ML base types

Most ML base types have close Java equivalents, so ML `int`s are represented by Java `int`s, and ML `string`s are represented as Java `String`s. There are a couple of small differences in the semantics of these types or their operations which have led us to diverge from the ML definition: integer arithmetic does not raise `Overflow` and for us the `Char` and `String` structures are the same as `WideChar` and `WideString` since Java is based on Unicode.

6.2 Products

Each distinct product type $\tau_i \equiv \tau_1 \times \cdots \times \tau_n$ is represented by a different class P_i whose fields have types $\tau_1, \ldots, \tau_n$ (see the class hierarchy in Figure 6). Because we monomorphise, representations can be shared by 'sorting' the fields by type, for example using the same class for the type $\mathsf{int} \times \mathsf{string}$ as for $\mathsf{string} \times \mathsf{int}$, but the current version of the compiler doesn't do this yet.

6.3 Sums

The natural 'object-oriented' view of a sum type $\tau_1 + \cdots + \tau_n$ is to represent the n summands as n subclasses of a single class and then to use method lookup in place of `case`. For example, one would represent lists by a class `List` with subclasses `Nil` (with no fields) and `Cons` (with a field for the head and a field for the tail). The `length` function would compile to a method whose implementation in `Nil` simply returns zero and whose implementation in `Cons` returns the successor of the result of invoking the method on the tail.

Whilst this technique is elegant, it is not necessarily efficient. Typical functional code would generate a large num-

ber of small methods; moreover it is necessary to pass in free variables of the bodies of `case` constructs as arguments to the methods.

If the JVM had a `classcase` bytecode then it would be possible to use this instead of method invocation. Unfortunately it does not: one must check the classes one at a time (using `instanceof`) and then cast down (using `checkcast`).

A variation on this idea is to store an integer tag in the superclass and to implement `case` constructs as a switch on the tag followed by a cast down to the appropriate subclass. We take this a stage further, using a single superclass S for all sum types with a subclass S_i for each *type* of summand. This reduces the number of classes required for summands; moreover, because only a single class is used to represent most sum types occurring in the source program, further sharing of representations is obtained in types constructed from sum types.

This 'universal sum' scheme is used in general, but for two special cases we use more efficient representations as described below.

- Enumeration types: for datatypes whose constructors are all nullary we use the primitive integer type.
- '1+' types: In Java, variables of *reference* type contain either a valid reference to an object or array, or the value `null`. We make use of this for types of the form $\langle\rangle + \tau$ where τ can itself be represented by a non-null class or array type. For example, the SML type `(int*int) option` is implemented using the same class as would be used for `int*int`, with `NONE` represented by `null`. The type `int list` is implemented by a single product class with fields for the head (an integer) and tail (the class itself) with `nil` represented by `null`.

  If τ is primitive then a product class is used to first 'wrap' the primitive value. What if τ is itself of the form $\langle\rangle + \tau'$? Then we create an additional dummy value of appropriate class type to represent the extra value in the type. For example, a value `SOME x` with the SML type `int list option` is represented in the same way as would be `x` of type `int list`, but `NONE` is an extra value created and stored in a global variable when the class is initialised.

6.4 Reference types

In general a reference type τ `ref` is simply represented by a unary product class. For references created at top-level (*i.e.* not inside functions) that are not used in a first-class way (assigned to and dereferenced but not compared for equality or passed around), we use static fields in a distinguished class G, in other words, global variables. In the future we hope to perform escape analysis in order to use local variables for reference types where possible.

6.5 Exceptions

In SML the type `exn` has a special status in that it is *extensible*. Exception declarations create fresh distinguishable exception constructors; in the operational semantics this is formalised by the creation of fresh names. However, for exceptions declared outside of functions there will be a fixed finite set of names that can be determined at compile-time. We exploit this by representing each such exception constructor by a separate class E_i that subclasses E, the class of 'ML exceptions'. In contrast to sums, we do not use the same class for exception constructors whose argument types are the same. ML's `handle` construct then fits better with the JVM try-catch construct where the class of the exception is used to determine a block of code to execute.

For *generative* exception declarations that appear inside functions, we generate a fresh integer count from a global variable and store this in a field in the exception constructor object. This field is tested against in exception handlers and case constructs.

Exceptions also give a nice anecdotal example of the sort of low-level Java-specific tweaking that we've found to be necessary in addition to high-level optimisations. In an early version of the compiler we noticed that certain programs which used exceptions as a control-flow mechanism ran hundreds of times more slowly than we would have expected. The problem was tracked down to a 'feature' of Java: whenever an exception is created, a complete stack trace is computed and stored in the object. The solution was simply to override the `fillInStackTrace` method in the ML exception class so that no stack trace is stored.

6.6 Functions

As mentioned earlier, functions in MIL are divided into locals, globals and closures. These are used as follows:

- Functions that only ever appear in tail application positions sharing a single 'continuation' can be compiled inline as basic blocks. Function application is compiled as a `goto` bytecode. Incidentally, this is one good reason for compiling to JVM bytecodes rather than Java source – the `goto` instruction is not available in Java. For example:

```
let
  val f = fn x => some_code
in
  if z<3 then f(z) else f(w+z)
end
```

  The body of `f` is simply compiled as a block of code and the two calls to `f` are compiled as jumps.

  It is sometimes even possible to transform a non-tail function application into a tail one. Consider the following fragment of ML:

```
let
  val f = fn x => some_code
  val y = if z<3 then f(z) else f(w+z)
in
  some_more_code
end
```

  Assuming that `f` does not appear in the expression `some_more_code`, then this 'continuation' can be moved into the definition of `f` and the calls to `f` implemented by jumps.
- Other functions appearing only in application positions are compiled as static Java methods in a distinguished class G. Function application is implemented by the `invokestatic` bytecode, unless it is a recursive tail call to itself, in which case `goto` is used.
- The remaining functions are used in a higher-order way and so must be compiled as closures. There are a number of ways that this can be achieved. The most obvious is to generate for each function type an abstract

class with an abstract *app* method, and then to subclass this for each closure of that type, storing the free variables as instance variables in the object. This is rather wasteful of classes, using one per function type and closure appearing in the program.

We currently use a different (and at first sight rather alarming) scheme. Instead of a single *app* method, we use different method names for different function types. There is a single superclass F of all functions, with dummy methods for each possible *app* method. Then closures with the same types of free variables but different function types share subclasses of F. It is even possible for the actual closure objects to be shared, if their free variables are the same but *app* methods are different. For example:

```
fun f (x:string) =
let
  val f1 = fn y:int => (x,y)
  val f2 = fn z:string => (x,z)
in
  ...
end
```

If closures are required for `f1` and `f2` then they can share the same closure class as their free variable types are the same but function types are different. Moreover, the *values* of their free variables are the same so the same object can be used for both, saving an allocation.

A simple flow analysis is used to decide how each function should be compiled. A more sophisticated flow analysis would not only allow us to identify more known functions, but would also refine our type-based partitioning of application methods, allowing more sharing of classes between closures.

7 BBC

The Basic Block Code (BBC) is a static single-assignment representation of the operations available in the Java virtual machine, which abstracts away from certain instruction selection details, including the distinction between stack and local variables. Normal form MIL (with all commuting conversions applied) is translated into BBC (which also includes some information about effects and which object fields are mutable) and the backend then orders and selects instructions to turn that into real bytecode. Currently the backend constructs a dependency DAG from the BBC and then works from the top down, using the stack where possible but (mostly) storing intermediate results in local variables where they are used more than once, or where ordering constraints make their immediate use on the stack impossible. After that, there is a pass in which local variable numbers are reassigned, so that the number of copies between basic blocks is minimised, combined with a standard register-colouring phase in which we try to minimise the total number of local variables used.

This scheme produces code which is respectable but far from optimal. Data passed between basic blocks is never left on the stack but is instead passed via local variables. Within basic blocks themselves, the JVM's stack is only used in a fairly simple-minded way. This causes too many local variables to be used and leads to code being somewhat larger than we would like. Heavy optimisation of the backend is probably not justified from the point of view of execution speed, given that the bytecodes will usually be recompiled by a JIT-compiling virtual machine which does its own independent mapping of stack locations and local variables to registers and memory, but we are also keen to reduce the size of the bytecodes. We are currently developing an improved backend which will make much more intelligent use of the stack.

Benchmark	MLJ	Moscow	SML/NJ
Nfib	3.2	0.5	0.4
Quicksort	5.7	1.3	0.7
Life	10.6	1.0	2.4
Knuth-Bendix	20.6	1.8	5.3
Mandelbrot	4.3	0.5	0.7
Boyer-Moore	51.8	5.0	8.0
FFT	13.7	1.5	2.4

Table 1: Compile times (seconds)

8 Current Status and Performance

MLJ 0.1 currently comprises about 60,000 lines of SML, written using SML/NJ version 110, plus the Basis library code. It is freely available over the web (at `http://research.persimmon.co.uk/mlj/`) as an SML/NJ heap image for Solaris, Win32, Linux and Digital Unix with the Basis code compiled in.

Although there is scope for further improvement, MLJ is already useful in real applications. Internal projects at Persimmon using MLJ include

- Writing functional SGML/XML stylesheets which can be downloaded into web browsers or run on servers. This involves a lot of interlanguage working, including with Javascript (using Netscape and Microsoft's browser-specific Java classes) and with third-party Java XML parsers.
- Implementing a graphical functional language for filtering and classifying events from web servers. This involves interworking with a third-party graph editor written in Java.

We have a number of nice demonstrations, including Paulson's Hal theorem prover for first-order logic [18] compiled with some third-party Java terminal code to produce an applet, several functional programs with graphical user interfaces and some which access an Oracle database via Java's JDBC API. The largest program which we have successfully compiled is around 12,000 lines (a compiler for ASN.1, producing C++).

8.1 Compile times

The compile times for a range of standard SML benchmark programs are shown in Table 1. All timings were taken on a 200MHz Pentium Pro with 64MB of RAM running Windows NT4.0. We have compared a recent (July 1998) internal version of MLJ (0.1e) with Moscow ML 1.42 and SML/NJ 110.

Benchmark	MLJ	Moscow	SML/NJ
Nfib	3.7	3.4	310
Quicksort	7.1	3.8	346
Life	11.4	5.9	337
Knuth-Bendix	24.6	9.8	360
Mandelbrot	4.2	3.7	316
Boyer-Moore	25.4	39.1	439
FFT	15.4	6.5	374

Table 2: Code size comparisons (kilobytes)

8.2 Code size

Table 2 lists the sizes of compiled code produced by the three compilers. To obtain a (roughly) fair comparison, each excludes the run-time system. For MLJ, the total size of the class files is given (so this excludes the Java interpreter required to run it). For Moscow ML, the size of the bytecode file alone is given (so this excludes the `camlrunm` interpreter required to run it). For SML/NJ, the size of the Windows heap image produced by `exportFn` is given (so this excludes the `run.x86-win32` run-time system required to run it).

8.3 Run times

Some preliminary benchmark times are shown in Table 3. All timings were performed on the same machine as the compilation benchmarks and we again compare MLJ 0.1e with Moscow ML 1.42 and SML/NJ 110. The run times do not include start-up time for the run-time system.

Four different Java implementations were used to run the code compiled by MLJ:

- `java NT`: the latest version (1.2 beta 3) of Sun's Java Development Kit running under Windows NT4.0 with Symantec's JIT (3.00.023(x)) enabled;
- `jview NT`: the latest version of Microsoft's JIT compiler (build 2613) running under Windows NT4.0;
- `kaffe Linux`: the latest version (0.9.2) of Tim Wilkinson's Kaffe JVM with JIT enabled, running under RedHat Linux 5.0;
- `java Linux`: Steve Byrne's 1.1.6v2 port of Sun's interpreting JVM running under RedHat Linux 5.0.

To illustrate the effect that initial heap size can have on performance, Sun's JVMs were tested twice: firstly with the default initial heap of 1MB and secondly with the heap starting at 30MB.

8.4 Interpretation of the results

As usual, the details of these small-program benchmark figures should be treated with some scepticism, but it's possible to make some broad generalisations.

The first thing to note is that MLJ compile times are very high (between 4 and 11 times slower than Moscow ML and between 4 and 8 times slower than SML/NJ), though it's worth reiterating that the recompile times, which are the important numbers for software development, are typically a third less than the total compile times given here. But it's hardly surprising that extensive functional rewriting of the whole program turns out to be a costly compilation technique – if SML/NJ is given a whole program as a single file, then its compile times are often higher than MLJ's. Our intermediate language certainly uses more space than a more traditional untyped lambda calculus would, firstly because we are carrying types around and secondly because the computational lambda calculus translations are inherently more verbose. This slows compilation by increasing heap turnover. The current parser also contributes to long compile times as it uses parser combinators rather than being table-driven.

Secondly, Java Virtual Machines vary widely in performance. A good JIT compiler produces significant speedups, but the current state of the art is that the fastest JITs also have bugs. Microsoft's Win32 JIT is generally quite fast but has a fundamental bug that sometimes causes operations to be unsoundly reordered. Luckily, we have been able to identify the problem sufficiently precisely to add a compiler option to produce slightly less efficient code which avoids the bug. The current version of Symantec's JIT is sometimes very good but has a number of serious bugs which often prevent it from running our code. These problems indicate a pragmatic (though we hope temporary!) drawback of 'clever' compilation of other languages to Java bytecodes – most Java compilers produce fairly naive, stylised bytecodes whereas MLJ produces rather more contorted bytecodes which, whilst perfectly legal according to the JVM specification, could not have been produced by any Java compiler. This tends to uncover bugs which have not have been found by JVM implementers who have only tested against the output of existing Java compilers.

In general, MLJ code run with a JIT compiler tends to have particularly good performance (even better than SML/NJ) on heavily numeric benchmarks such as Nfib, Mandelbrot and FFT. This is unsurprising, as our monomorphisation should allow such code to be easily translated by a JIT into much the same code as would be generated by a naive C compiler. More typical functional code which does a lot of heap allocation (e.g. Quicksort, Boyer-Moore and Life) tends to run rather more slowly, showing that storage management in JVMs is still fairly poor (we suspect that the fact that increasing the initial heap size makes a significant difference to Quicksort, Knuth-Bendix and Boyer-Moore but not to Life when running on the Sun/Symantec JVMs indicates inefficient heap expansion rather than just slow garbage collection *per se*). The comparison with SML/NJ on the Life benchmark is particularly interesting – taking the benchmark as originally written, our whole program optimisation allows us to specialise representations, including uses of polymorphic equality, and run up to 3 times faster than SML/NJ. However, when the program is constrained with a minimal signature, SML/NJ can also specialise and runs nearly 4 times faster than the best MLJ can manage. The particularly poor performance of Microsoft's JIT (and unimpressive performance of the others) on the Knuth-Bendix benchmark seems to be due to the fact that as well as doing a good deal of allocation, it makes heavy use of exceptions.

The code produced by MLJ is impressively compact but not quite as small as that produced by Moscow ML. Although Moscow has the advantage of a bytecode specifically designed for ML, we expect to be able to narrow the gap in future. We have already mentioned ongoing improvements to our backend, but just as significant is the non-trivial space overhead associated with the fairly large number of distinct Java classes produced by MLJ. For example, the Knuth-Bendix benchmark produces a total of 42 classes and over

Benchmark	MLJ						Moscow	SML/NJ
	`jview` NT	`java` NT	`java` NT 30MB	`kaffe` Linux	`Java` Linux	`Java` Linux 30MB		
Nfib	0.8	0.9	0.9	1.7	5.1	5.1	8.2	1.3
Quicksort	†8.5	‡25.1	‡17.7	109.1	35.7	18.2	21.8	0.9
Life	7.0	7.1	6.3	16.3	32.3	31.9	38.3	∗18.9
Knuth-Bendix	†82.1	37.0	10.7	426.8	63.0	26.9	10.0	2.4
Mandelbrot	32.7	‡217.9	‡217.9	167.4	217.2	217.4	322.7	41.9
Boyer-Moore	4.2	‡6.8	0.9	37.3	8.3	3.3	2.1	0.6
FFT	15.5	11.8	12.3	28.8	71.3	71.8	441.6	⋆28.7

† requires compilation with MLJ's `microsoftbug` switch set to avoid bug in the Microsoft JIT

‡ program crashed due to a bug in the Symantec JIT; timing is with JIT disabled

⋆ SML/NJ gave incorrect results

∗ timing improves to 1.7 seconds when top-level structure is constrained by minimal signature

Table 3: Run times (seconds)

6K of the total size of 24.6K is taken up by the product, sum, exception and F classes, each of which contains essentially no 'real' code. There is certainly scope for improving our representation choices to decrease code size still further.

9 Conclusions and Further Work

MLJ is a very useful tool: a compiler for a popular functional language which produces compact, highly portable code with reasonable performance and which has unusually powerful and straightforward access to a large collection of foreign libraries and components. But the reasonable performance has only been achieved at the price of high compile times and a limitation on the size of programs which may reasonably be compiled. Our decision to do whole-program optimisation is certainly controversial, so it seems worth trying to give some explicit justification:

- Most importantly, because of the relative inefficiency of current JVMs and the difficulty of mapping ML to Java bytecodes, it was simply the only way to achieve what we considered to be adequate performance.
- The limitation on program size is just not a problem for many real applications. The number of SML programs which are more than, say, 15,000 lines long is actually rather small, as one can do an awful lot in that much SML. We have had it suggested to us that a compiler which cannot compile itself is useless, but this is clearly nonsense.
- Trends towards component architectures, interlanguage development, dynamic linking and distributed systems mean that the large monolithic application is becoming less common. Of course, component boundaries reintroduce the problems of separate compilation in a worse form, but that's all the more reason to compile each component as well as possible.
- Completely separate compilation at the granularity of modules introduced for software engineering purposes is an anachronism for which high-level languages can pay dearly, as the earlier discussion of SML/NJ's performance on the Life benchmark indicates (more realistically, we have ourselves doubled the speed of parts of MLJ simply by manually demodularising the code). There is a whole range of approaches between completely naive whole-program compilation and simple-minded separate compilation and, whilst the optimum lies somewhere in the middle, the two extremes are much the easiest for the compiler writer. As MLJ develops, caching more information about each module and sacrificing some rewrites to be able to handle larger programs, and other compilers add ever more complex intermodule optimisations, we expect them to come much closer together.

It is interesting to compare MLJ with Wadler and Odersky's Pizza [17]. We have started with a standard functional language and added extensions to support interlanguage working with Java, whereas Pizza starts with Java and adds some 'functional' features, such as pattern matching and parameterised types. We are aware of two other attempts to compile SML into Java, one by Bertelsen [6], based on Moscow ML, and the other by Walton at Edinburgh. Both of these use uniform representations and do not perform anything like the same level of optimisation as MLJ. Wakeling has also compiled Haskell into Java bytecode, with 'disappointing' results (large code and performance considerably slower than the Hugs interpreter) [25].

The other main attempt to compile a mainly functional language into Java bytecode is Bothner's Kawa compiler for Scheme [8]. Kawa has an interactive top-level loop and compiles bytecodes dynamically, but has to use uniform representations. Some informal tests indicate that Kawa typically runs an order of magnitude more slowly than MLJ and tends to run out of memory or stack space much earlier than MLJ.

We are currently developing concurrency extensions for MLJ which are built on top of Java's built-in threads and, looking further into the future, are thinking about the possibility of taking advantage of Java's remote method invocation infrastructure to develop distributed and mobile applications in ML.

We have reason to believe that JVMs which do tail call elimination may appear soon, which would remove one of the other significant limitations of MLJ – although we already compile most simple loops into jumps, the lack of more general tail call optimisation does make some programs run out of stack space on reasonable-sized inputs. If tail call optimisation is not done for us, then we will reluctantly have to consider selective use of two techniques: placing some functions in the same method, and the 'tiny interpreter' technique used in the Glasgow Haskell compiler [10]. Naive use of of these techniques would cause a significant worsening of code size and speed, so we would base our decisions on a more sophisticated flow analysis, which would also allow us to improve most of the other transformations [21, 16]. This would appear to point to even longer compilation times, but we hope that this can be avoided by improving the representation of our intermediate language. The compiler currently spends a vast amount of time (and memory) performing trivial rewrites on the MIL term. Many of these rewrites are commuting conversions, which would simply disappear if a suitable graph representation were used instead. If we were also to rewrite destructively, then we should be able to obtain further compiler speedups. A further minor improvement which we need to make is to ensure that MLJ never generates methods which exceed the JVM's 64K byte limit. This has so far only happened to us on one unusual program, and we do not anticipate any great difficulty in modifying the code generator to prevent it happening.

Our use of a monadic intermediate language is particularly novel. Whilst we would not claim that this allows us to perform any optimisations which could not be achieved by more ad hoc methods, we have found it to be a powerful and elegant framework for structuring the compiler. In general, we would strongly advocate a principled use of type theoretic and semantic ideas in compiler implementation.

One of the good things about compiling to Java bytecodes is that there is a large ongoing effort to develop better faster JVMs which we can take advantage of for free. Early information from Sun about their next generation of JVMs indicates that allocation and collection will be improved significantly, possibly by a factor of 4. That could make MLJ's runtime performance competitive with native compilers even if we made no further improvements ourselves.

References

[1] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5), September 1997.

[3] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, second edition, 1998.

[4] P. N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, University of Cambridge Computer Laboratory, August 1993. Technical Report 309.

[5] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 1998. To appear.

[6] P. Bertelsen. Compiling SML to Java bytecode. Master's thesis, Dept. of Information Technology, Technical Univ. of Denmark, January 1998.

[7] M. Blume. CM: A compilation manager for SML/NJ. Technical report, 1995. Part of SML/NJ documentation.

[8] P. Bothner. Kawa – compiling dynamic languages to the Java VM. In *USENIX Conference*, June 1998. Compiler and paper available from `http://www.cygnus.com/~bothner/kawa.html`.

[9] F. Henglein and J. Jørgensen. Formally optimal boxing. In *ACM Symposium on Principles of Programming Languages*, pages 213–226, 1994.

[10] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, pages 127–202, April 1992.

[11] S. L. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *ACM Symposium on Principles of Programming Languages*, January 1998.

[12] J. Jørgensen. A calculus for boxing analysis of polymorphically typed languages. Technical Report 96/28, DIKU, University of Copenhagen, May 1996.

[13] X. Leroy. Unboxed objects and polymorphic typing. In *19th Annual ACM Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[14] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Mass., 1997.

[15] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[16] C. Mossin. Flow analysis of typed higher-order programs. Technical Report 97/1, DIKU, University of Copenhagen, 1997.

[17] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages*, January 1997.

[18] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, 1996.

[19] Z. Shao. An overview of the FLINT/ML compiler. In *ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, June 1997.

[20] Z. Shao. Typed cross-module compilation. Technical Report YALEU/DCS/TR-1126, Department of Computer Science, Yale University, July 1997.

[21] O. Shivers. *Control-flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[22] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[23] A. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Workshop on Types in Compilation*, March 1998.

[24] P. Wadler. The marriage of effects and monads. In *3rd ACM SIGPLAN Conference on Functional Programming*, September 1998. (this volume).

[25] D. Wakeling. VSD: A Haskell to Java virtual machine code compiler. In *9th International Workshop on Implementation of Functional Languages*, September 1997.

[26] S. Weeks. A whole-program optimizing compiler for Standard ML. Technical report, NEC Research Institute, November 1997. Available from: `http://www.neci.nj.nec.com/homepages/sweeks/smlc/`.

A Sample output

The code below implements the quicksort algorithm for integer lists.

```
fun quick xs =
let
  fun quicker (xs, ys) =
  case xs of
    [] => ys
  | [x] => x::ys
  | a::bs =>
    let
      fun partition (left,right,[]) =
          quicker (left, a::quicker (right, ys))

        | partition (left,right, x::xs) =
          if x <= a
          then partition (x::left, right, xs)
          else partition (left, x::right, xs)
    in
      partition([],[],bs)
    end
in
  quicker (xs, [])
end
```

The internal function `quicker` compiles to the following static method:

```
Method Ra b(Ra, Ra)
   0 goto 31
   3 new #22 <Class Ra>
   6 dup
   7 iload_2
   8 aload_0
   9 invokespecial #38 <Method Ra(int,Ra)>
  12 astore_0
  13 goto 55
  16 new #22 <Class Ra>
  19 dup
  20 iload 4
  22 aload_3
  23 aload_1
  24 invokestatic #39 <Method Ra b(Ra, Ra)>
  27 invokespecial #38 <Method Ra(int,Ra)>
  30 astore_1
  31 aload_0
  32 ifnull 105
  35 aload_0
  36 getfield #35 <Field Ra b>
  39 dup
  40 astore 5
  42 ifnull 92
  45 aload_0
  46 getfield #37 <Field int a>
  49 istore 4
  51 aconst_null
  52 astore_3
  53 aconst_null
  54 astore_0
  55 aload 5
  57 ifnull 16
  60 aload 5
  62 getfield #37 <Field int a>
  65 dup
  66 istore_2
  67 iload 4
  69 aload 5
  71 getfield #35 <Field Ra b>
  74 astore 5
  76 if_icmple 3
  79 new #22 <Class Ra>
  82 dup
  83 iload_2
  84 aload_3
  85 invokespecial #38 <Method Ra(int,Ra)>
  88 astore_3
  89 goto 55
  92 new #22 <Class Ra>
  95 dup
  96 aload_0
  97 getfield #37 <Field int a>
 100 aload_1
 101 invokespecial #38 <Method Ra(int,Ra)>
 104 areturn
 105 aload_1
 106 areturn
```

This program illustrates some of the code transformations performed by MLJ.

Integer lists have been represented by the class `Ra`, with `nil` represented by `null` and `x::xs` represented by an instance of `Ra` with `x` stored in field `a` and `xs` in field `b`.

Notice how the calls to `partition` and the first call to `quicker` have been implemented by `goto` bytecodes. The tuples have been removed: the triple passed to `partition` has vanished, and the function `quicker` expecting a pair has been transformed into a method with two arguments.