Compiling Mercury to the .NET Common Language Runtime

Tyson Dowd, Fergus Henderson

Department of Computer Science
and Software Engineering
The University of Melbourne
{trd, fjh}@cs.mu.oz.au

Peter Ross

Mission Critical, Belgium
peter.ross@miscrit.be

Abstract

The .NET Common Language Runtime (CLR) offers a new opportunity to experiment with multi-language interoperation, and provides a relatively rare chance to explore deep interoperation of a wide range of programming language paradigms. This article describes how the logic/functional programming language Mercury is compiled to the CLR. We describe the problems we have encountered with generating code for the CLR, give some preliminary benchmark results, and suggest some possible improvements to the CLR regarding separate compilation, verifiability, tail calls, and efficiency.

1 Introduction

We have been researching language interoperability as part of our research on the declarative logic/functional programming language Mercury [9].

Many programming language researchers (including ourselves) have argued that if we are to see widespread adoption of new programming languages, they must include comprehensive interfaces with existing languages and systems.

Microsoft's .NET Common Language Runtime (CLR) [11] is a new platform that may help make such interfacing easier. The CLR is intended to support a range of programming languages using a common data representation, garbage collector, calling convention, and exception handling mechanism, and comes with a large set of class libraries. This provides a high-level base for interoperation, leaving us with just the built in assumptions and natural mismatches in language semantics and features to tackle.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The first step towards supporting language interoperability between Mercury and other languages on the .NET CLR is to implement a compiler for Mercury that generates code for the CLR. That step is what this paper describes.

2 Preliminaries

2.1 The .NET Common Language Runtime

The .NET CLR is a new virtual machine developed by Microsoft. The system is similar to the Java virtual machine in that it is based around an object-oriented type system, a bytecode based instruction format, verifiable code and a builtin security system. It differs in that it is intended to be a replacement runtime system for Microsoft's stable of programming languages (Visual Basic, Visual C++, JScript and $C\sharp$), whereas the JVM is intended to just run Java [1]. In addition, the .NET CLR has some explicit support for languages outside the inner circle, such as a tail-call instruction (see Section 3.4).

The Mercury compiler targets the .NET CLR by generating classes which define methods using instructions in Common Intermediate Language (CIL or sometimes simply IL) in an assembly language. A CIL assembler turns a textual representation of CIL into .NET byte code in the appropriate file formats.

The .NET CLR implementation loads and optionally verifies code as it is required by the running program. Native code is generated from the CIL by a just-in-time compiler. It is also possible to mix native code with CIL, if verification is not required.

The CLR's type system is more expressive than the JVM, subsuming all the familiar Java types — distinguishing between value classes (structs) and reference classes (heap-allocated objects with identity), and providing unsigned integer types, safe ("managed") pointers for pass-by-reference, and traditional unsafe ("unmanaged") pointers for code that doesn't need verifiability.

The system also defines a subset of the allowable types, names and other constructs which can appear in component interfaces, called the Common Language Specification (CLS), which is intended to be a minimal standard for interoperation between different programming languages and tools.

The CLR allows reflection on all the names, types, signatures and parameters of loaded modules. Attributes may be attached to types, modules, parameters and signatures by users, compilers or tools in a fairly general way, which allows new interoperation standards to be built on top of the existing runtime without requiring explicit support.

---

[1] Although there have certainly been many efforts to run other languages on the JVM: see `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`

2.2 Mercury

Mercury is a pure logic/functional programming language intended for general purpose large-scale programming.

We assume familiarity with the basic concepts of logic and functional programming. We do not assume familiarity with aspects of Mercury beyond the following.

- Mercury is strongly typed. It has a Hindley-Milner style type system, similar to that of Haskell and ML.
- Mercury supports functions as well as predicates. Every predicate and every function has one or more *modes*, which say, for every argument or function value, whether it is input or output. (The mode system is more sophisticated than that, but that is all that matters for this paper.) We call each mode of a predicate or function a *procedure*.
- Mercury associates with every procedure a *determinism*, which expresses upper and lower bounds on the number of solutions the procedure can have. Procedures that are guaranteed to have exactly one solution have determinism *det*. Procedures that may have one solution for one set of inputs and zero solutions for other inputs have determinism *semidet*. Procedures that may have any number of solutions have determinism *nondet*; nondet procedures use backtracking search.
- Mercury's `main` must be *det*. Therefore all code must eventually interface with deterministic code. A *commit* is generated when non-deterministic code is called from a context that expects only one solution; the commit will prune the nondet backtracking search to a single solution.

These concepts are introduced in the Mercury tutorial [1] and described in detail in the Mercury language reference manual [9] and elsewhere [10].

2.3 Related Work

Many languages have been ported to the JVM to achieve interoperability with Java. Of these ports, the MLj compiler [2,3] is probably the most closely related work to this paper, since ML has a similar type system and execution algorithm to Mercury. However MLj is a whole-program compiler, and is not a logic programming language, so we encountered several issues that do not arise in the MLj system.

Several Prolog (JProlog, NetProlog) and logic programming languages (Jinni [13]) have targeted the JVM; however, because of the strong static type, mode and determinism information in Mercury, the execution algorithm used for Mercury has little in common with most logic languages. These Prolog implementations do provide interesting ideas about interfacing logic programming concepts with Java.

Little published work is available as yet on systems that interoperate on

.NET.

3 Operation

3.1 Compiler front-end and intermediate representation

The original compiler for Mercury [8,12] compiled to very low-level code, using techniques that were not appropriate for compiling to the .NET CLR.

Recently we developed a new code-generator for Mercury that generates higher-level code, which is more suitable for high-level target languages like CIL. The new code-generator is based around a new intermediate data structure, modelled on a high-level abstract imperative language named the Medium Level Data Structure, or MLDS. There are several different back-ends that transform the MLDS into actual target code (C, Java, native assembler or CIL).

The MLDS consists of constructs such as assignments, calls, simple expressions (including taking addresses of variables and dereferencing pointers), allocation and deallocation. The type system contains the usual primitive types, as well as classes, structures, arrays, interfaces, enumerations, and function pointers. There is support for instance and static members, virtual functions, and access restrictions (public, private, protected, or similar). These are common elements in almost all modern imperative programming languages, hence it is relatively easy to target imperative programming languages such as C or Java.

The MLDS code generator translates Mercury code into relatively simple imperative programming constructs. The details of that translation are discussed in depth in a different paper [7], so we won't discuss them here. The aim of this paper is to address the issues which are specific to compiling MLDS to the .NET CLR.

3.2 Modules and static data

Each Mercury module becomes a .NET namespace containing a class called '`mercury_code`'. Here's the CIL code that the Mercury compiler generates for an empty Mercury module. You'll note that the CIL code looks a bit like a cross between a high-level object oriented language and assembler. Directives like '`.namespace`' and '`.class`' introduce classes, methods, namespaces, fields, etc. Inside each method, the code contains bytecode assembler instructions. For clarity, we've added equivalent *C*♯ code in comments.

```
.namespace mymodule {
  .class public mercury_code {
    .field public static bool rtti_initialized
    .method static default  void .cctor() {
        // if (rtti_initialized) return;
        // rtti_initialized = true;
```

```
        // mercury.builtin.mercury_code::'.cctor'();
        ldsfld bool mymodule.mercury_code::rtti_initialized
        brfalse l1
        ret
    l1:
        ldc.i4.1
        stsfld bool mymodule.mercury_code::rtti_initialized
        call void [mercury]mercury.builtin.mercury_code::.cctor()
        ret
} } }
```

The '`mercury_code`' module contains a class constructor (called '`.cctor`') which initialises the static data required by this Mercury module. The Mercury compiler generates static data for three purposes: (a) as an optimization, Mercury terms that don't contain variables are constructed statically rather than each time they are referenced; (b) static data structures are needed for the run-time type information needed to handle Mercury's parametric polymorphism and type classes (see 3.8.4); and (c) the Mercury compiler generates static hash tables for indexing string switches.

The .NET CLR does not allow static objects to be constructed at compile time or link time [2], so we need to initialize them at runtime. Currently this means initialising the local run-time type information (RTTI), then initialising RTTI in other modules on which the current module depends, and finally filling in any cross-module references in the RTTI data structures. Note that in this sample program no RTTI is initialised, but the class constructor of other modules is called. We keep track of whether we have already initialised the RTTI in a module with a static boolean flag. This allows us to avoid looping in the case of mutually dependent modules.

The Mercury standard library is quite large, and contains many types, and so the Mercury compiler generates a considerable amount of RTTI for it. Initializing these static data structure at run time may have a significant performance impact, particularly on program start-up times.

An alternative approach would be to initialize these structures lazily, delaying the initialization of each data structure until it is first used. That would lead to better start-up times, but only at the expense of slowing down long-running programs, due to the additional overhead of checking whether the data structures are already initialized on each access. We have not implemented that approach.

[2] There is some support for static byte arrays, but since the .NET CLR is intended to support multiple hardware architectures, object layout is not known at compile time or link time, so .NET objects can only be created at runtime. To support statically initialized data in a verifiable way, the JIT would have to optimize the initialization into a static allocation of data, and cache such data on disk.

3.3 Procedures

Each mode of a function or predicate is called a procedure. We map each procedure to a .NET class member function that is a static member of the '`mercury_code`' class.

Mercury predicates become static .NET functions with '`void`' return values. Mercury functions become static .NET functions with a return value corresponding to the function output. Functions whose return value is not an output (possible, but not common) are treated as predicates.

Because Mercury allows functions and predicates to have the same name and arity, as well as allowing multiple modes of the same function or predicate, we have to do some name mangling. We cannot rely on .NET's overloading to resolve overloading for us, as there are some cases where Mercury will map different predicates or functions to the same .NET method signature and name. We use a static scheme for mangling procedure names so that no collisions occur.

In practice, apart from appending the arity to each procedure, the amount of mangling is quite small.

3.4 Tail Calls

Recursion is the natural looping construct in logic programming languages, so it is used extensively. Tail call elimination very important, both for optimization and to ensure bounded stack space usage for tail-recursive loops.

The .NET CLR provides a `tail.` instruction prefix which when coupled with a call will perform a call that uses constant stack space. Typically this is implemented by re-using the stack-frame of the caller for the callee. The `tail.` prefix must precede a `call`, `calli` or `callvirt` instruction that is followed by a `ret`.

For directly recursive tail calls, where the caller is the same procedure as the callee, we implement tail call elimination by generating code that uses a branch instruction to loop back to the start of the procedure. For other tail calls, we use the CLR's `tail.` instruction prefix to implement tail call elimination. However, as explained below, there are some problems with verifiability when using the CLR's tall call support to call procedures with multiple output parameters.

3.5 Output Parameters

Mercury supports procedures with multiple output parameters. This can be difficult to map to some platforms, because it is common for programming languages and virtual machines to only allow a single return value.

Fortunately, however, the .NET CLR provides a mechanism, *by-ref* parameters, for passing parameters by reference. .NET by-ref parameters are verifiable and safe; the verifier prevents you from generating by-refs that are dangling (i.e. you can only pass them into calls, you can't return them or store them in static variables or on the heap).

So we implement Mercury's multiple output parameters using by-refs.

```
:- func add(int, int) = int.
:- mode add(in, in) = out is det.
:- mode add(out, in) = in is det.
add(X, Y) = X + Y.
```

Becomes:

```
.method static default int32 add_3(int32 HeadVar__1_1,
                int32 HeadVar__2_2) {
        .maxstack 2
        // return HeadVar__1_1 + HeadVar__2_2;
        ldarg HeadVar__1_1
        ldarg HeadVar__2_2
        add
        ret
}

.method static default void add_3_f_m1(int32& HeadVar__1_1,
        int32 HeadVar__2_2, int32 HeadVar__3_3) {
        .maxstack 3
        // *HeadVar__1_1 = HeadVar__3_3 - HeadVar__2_2;
        ldarg   HeadVar__1_1
        ldarg   HeadVar__3_3
        ldarg   HeadVar__2_2
        sub
        stind.i4        // "indirect store of 4-byte integer"
        ret
}
```

The CLR and associated tools also have a convention aimed at solving the dilemma of in/out versus out. There is an '`out`' attribute that can be placed on a parameter, which can be used by other compilers and tools to indicate that a by-ref parameter is output only. This corresponds nicely with Mercury's '`out`' mode, and is an example of how attributes can be used to improve the interoperability of the system without requiring changes to the underlying system.

Unfortunately, the verifier doesn't check and therefore cannot rely upon the '`out`' attribute; CIL code will not be verifiable unless it initializes parameters that are passed by reference before passing them (regardless of whether the reference is just being used for an output parameter).

There is, however, a more serious problem with the current CLR verifier's treatment of by-refs. The verifier does not allow tail call instructions to use by-ref parameters, presumably because the analysis required to ensure no references refer to the local stack frame has not yet been implemented. This is an unfortunate problem, requiring us to give up verifiability, tail-calls, or by-ref parameters, so we hope it will be remedied in future versions.

In our current implementation we normally generate unverifiable code with proper tail-call elimination, but we also provide an option to generate verifiable code by not doing tail-call elimination for procedures with '`out`' mode parameters.

Another possibility would be to return multiple value in structures (value types). However, there are several drawbacks to this approach. The first is that this approach is not quite as natural a mapping as by-ref parameters, which would make interoperability more difficult. Programmers trying to call Mercury code from another language might have difficulty understanding the documentation for the Mercury code, because correspondence between the Mercury code and its CLR interface would not be as direct, since the CLR interface would be returning a value type where the Mercury code was using '`out`' mode parameters.

A second drawback is that this approach can make tail call elimination more difficult; when return values must be moved from the structure to their intended destinations after a call, tail-call elimination is inhibited.

A third drawback of this approach is that it would require defining a new value type for each different set of return types. That might increase the size of the generated code considerably. (Previous experience with the JVM suggests that this can be a real problem.) It would be possible to reduce the number of types needed by using tuple structures whose fields are the generic "object" type, but then additional code would be needed for boxing values before inserting them into the tuples and for downcasting and unboxing them when retrieving from the tuples, and this would have very significant costs for execution time.

3.6 Non-determinism

Non-deterministic procedures are represented in the MLDS as a set of nested functions, one for each non-det goal. Each nested function *calls a continuation* (to the next non-det goal) if it *succeeds* (finds a solution), and *returns* if it *fails* (finds no solution). The nested functions share an environment (the local variables and actual parameters of the parent function), which represents the state of the saved values required to continue searching if a particular branch happens to fail.

All non-deterministic code will eventually interface to deterministic code (recall that Mercury's `main` must be deterministic). Hence there will be a sequence of (now unnecessary) continuation calls on the stack representing the search space.

If the non-deterministic code *fails*, it will eventually *return* to some semi-deterministic context (for example the condition of an if-then-else), where the failure will be handled.

Non-deterministic code *succeeds* to deterministic code by *commit*, which removes the continuation stack frames (see Section 3.7).

Since .NET does not support nested functions, we eliminate them by an MLDS-to-MLDS transformation that hoists them up to the top level. The shared environment variables are put into environment structures which are explicitly passed when continuations are called. These environment structures are turned into value classes in the .NET back-end. We use function pointers to implement continuation passing, using the `ldftn` and `calli` instructions to load function pointers and call through them.

The transformation to eliminate nested functions works smoothly in the C back-end, but there is a major complication when targeting the .NET CLR. Since Mercury uses by-ref parameters to represent output parameters, and the environment structures contain the actual parameters of the parent function, the environment structures contain by-refs if there are output parameters. However, on .NET one cannot put by-ref parameters inside structures; by-refs can only be stack variables or function parameters, so as to prevent the creation of dangling by-ref parameters that live on the heap, or are returned through a structure.

It would be a useful change to the .NET CLR to allow value classes to contains by-ref parameters, so long as value classes containing by-refs abide by the same rules for validity and verifiability that already exist for by-refs themselves. This extension may be useful for other programming languages that support by-ref parameter passing and nested functions.

To work around this problem, we use an approach that was suggested to us by Erik Meijer. We generate a different calling convention for non-deterministic code, which we call *non-det copy out*. Instead of passing output arguments by reference, and simply assigning to the output arguments whenever we produce an output binding, we generate local variables for locally produced outputs, and pass them to the continuation (and this continuation passes them to the next, and so on). When non-deterministic code ends (at a commit) we copy the set of output arguments from the final continuation into the (by-ref) output arguments.

Unfortunately that work-around still does not suffice. The problem is that environment structures may need to contain references to other environment structures (e.g. the environment for the caller) whose exact type is not known at compile time. The .NET CLR type system does provide a `refany` type (a.k.a. `System.TypedRef`) which can be used for safe references to values of unknown types; it holds both a pointer and a type, and there's a dynamically checked operation for converting a `refany` to a reference to a specific type. However, `refany` is subject to similar restrictions to by-refs. The copy-in copy-out work-around doesn't work for `refany`, because the size of the type that a `refany` value refers to isn't known at compile time.

For unverifiable code we can solve this by just using an unmanaged pointer type instead of `refany`, and using unchecked coercions to convert this to a specific by-ref type. But if we want verifiable code, the only alternative possible is to allocate the environment structures on the heap, using class types, rather than on the stack, using value types; this is likely to be less efficient.

Currently our compiler by default generates unverifiable code that allocates the environments for nondet Mercury procedures on the stack and uses unmanaged pointers to refer to them. However, we also provide an option for generating code which allocates the environments on the heap.

Unfortunately the code that we generate for nondeterministic Mercury procedures is still unverifiable, even if this option is enabled, because of our use of function pointers (`ldftn` and `calli`), which are unverifiable in the current .NET CLR. To remedy that, we plan to move to using the .NET delegates, which provide similar functionality to function pointers, and are verifiable, but carry the overhead of an object instead of a value.

3.7 Commits

The commit mechanism in the Mercury MLDS back-end requires some sort of stack unwinding mechanism to return to a previous point in the computation.

The Mercury compiler's MLDS code generator represents commits using a pair of special MLDS constructs: `try_commit` encapsulates a block of code which might do a commit, and `do_commit` unwinds the stack back to the matching `try_commit`.

In the .NET back-end, these are implemented using exceptions. The MLDS '`try_commit`' becomes an exception handling *try block* and a *catch block*, and '`do_commit`' simply loads a special Mercury commit type and throws it as an exception, which will be caught by the catch block of the nearest `try_commit`.

3.8 Data Representation

3.8.1 Primitive types

Implementing Mercury's primitives types on the CLR is straight-forward: we map each of Mercury's primitive types to the corresponding CLR type. Mercury `int` becomes CLR `int32` (32-bit signed integers), Mercury `char` becomes CLR `char` (16-bit Unicode characters), Mercury `float` becomes CLR `float64` (64-bit IEEE float), and Mercury `string` becomes the CLR `System.String` class.

3.8.2 Arrays

Mercury arrays are also represented using CLR arrays. For any Mercury type MT, where MT is not a type variable, the Mercury type `array(`MT`)` becomes the CLR type CT`[]`, where CT is the CLR type corresponding to MT. Polymorphically typed Mercury arrays need to be treated specially; if MT is a type variable then the Mercury type `array(`MT`)` is mapped to the

CLR class `System.Array`, which is the base class that is inherited by all CLR array types.

3.8.3 Discriminated unions

For Mercury's discriminated union types (also known as algebraic data types), our current implementation uses a representation which is similar to the representation described in [6]. Every Mercury discriminated union type is represented as an array of objects, using the CLR type `System.Object[]`. The first element of the array is an integer tag used to distinguish between different constructors for discriminated union types.

This representation is unappealing, having significant drawbacks for both efficiency and interoperability, and was used mainly for historical reasons.

We've also been working on a higher-level representation, where each Mercury discriminated union type is mapped to a CLR abstract base class, and each constructor in a discriminated union type is mapped to a CLR class that derives from the abstract base class for that discriminated union type. However, this is not yet fully implemented. One difficulty with this representation is that some tricky issues arise with abstract data types. These are discussed in Section 3.8.6.

3.8.4 Polymorphic types

The handling of parametric polymorphism and type classes is very similar to the way it is done in the original Mercury compiler. Polymorphically typed Mercury variables are represented as `System.Object`, the root class in the CLR class hierarchy; the compiler inserts code to box and unbox value types such as `int32`, `char`, and `float64` when converting them to/from `System.Object`. For polymorphic procedures, the compiler inserts extra parameters that hold run-time type information (RTTI) and/or type class dictionaries (tables of class methods). The run-time type information is needed for Mercury's RTTI features, which are used for purposes such as optional dynamic typing and serialization/deserialization. The details are discussed in our previous work [6].

Note that although the CLR has extensive run-time type information and reflection facilities, we can't use the CLR's facilities to support Mercury's RTTI features; the CLR types don't have enough information. The CLR type system doesn't support parametric polymorphism, so information about the values of type parameters is lost when mapping Mercury types to CLR types. Different Mercury types, such as for example `list(string)` and `list(int)`, will map to the same CLR type.

3.8.5 Higher-order types

Closures are handled using environment structures and function pointers. This has some drawbacks for interoperability and verifiability and, as with our treatment of continuations, we plan to eventually move to using delegates instead.

3.8.6 Abstract Data Types

Mercury allows modules to define abstract types that are implemented as type synonyms. From outside the module, such a type is considered a distinct new abstract type, but from inside the module, the type is considered as equivalent to the synonym type.

Unfortunately there is no direct support for this in .NET. We need to generate specific type names and type specific code for referencing and manipulating abstract data types. So we need some way of handling abstract equivalence types for the .NET back-end [3].

The cleanest solution to this problem would be for the .NET CLR to support type synonyms, and support generic code for manipulating types. The type references and generic code could be expanded into type specific references and code at runtime.

There are several solutions we can use in the Mercury compiler to avoid this problem, none of which are completely satisfactory.

- Map everything to a single type, such as '`Object`'.

  This is very bad for interoperability (which is the whole point of this back-end), and probably very bad for performance too. Excluding value types such as '`int`' and '`float`' from being mapped to '`Object`' might improve performance and interoperability, but abstract equivalence types that are equivalent to '`int`' or '`float`' will need to be handled somehow.

- Map ADTs to reference classes externally, and cast to equivalence internally.

  This improves interoperability significantly, but at a steep cost in performance. Converting from a reference type to a value type internally might seem relatively painless, but if the ADT is nested inside a data structure (for example, the elements of a list or array) we may need to convert every element as part of the cast operation, which can be arbitrarily expensive.

- Treat ADTs as concrete types rather than abstract types.

  This solution gives good interoperability and efficiency, but it fundamentally abuses the notion of abstraction. While we can (and should, for type-checking purposes) ensure the Mercury compiler hides the definition of the data type from the user, compilers for other languages might not.

  Furthermore, this approach has unfortunate consequences for versioning. Because the Mercury compiler does separate compilation, this solution fundamentally changes the compilation model. With this approach, a module must be recompiled if the *implementation* of an abstract data type it imports changes.

Currently we use a variant of the first option. Many languages that support abstract data types could run afoul of this issue, depending upon their data

[3] the JVM and hence the Mercury compiler's Java backend suffers from the same problem

Table 1
Preliminary benchmark results

Language	compiler	target	`true`	`tak`	`queens`
Mercury	`mmc -s asm_fast.gc`	native	0.07	31.4	2.24
Mercury	`mmc -s asm_fast`	native	0.07	31.4	0.86
Mercury	`mmc -s hlc.gc`	native	0.07	34.2	1.76
Mercury	`mmc -s il`	CLR	N/A	N/A	9.28
Mercury	`mmc -s ilc`	CLR	1.80	34.6	14.4
C	`cl`	native	0.05	35.5	N/A
C	`gcc -O3 -fomit-frame-pointer`	native	0.07	35.6	N/A
C♯	`csc /o`	CLR	0.16	40.6	N/A

representation and compilation model.

4 Benchmarks

All benchmarks were carried out a 366 MHz Celeron, with 128 kb L2 cache and 256 Mb RAM, using Microsoft .NET Beta 2 and Cygwin gcc 2.95.2-6 on Windows 2000. Each benchmark program was run 6-8 times in succession, to fully prime the cache, with the result being the fastest time recorded.

We compared five different Mercury compiler back-ends — the one described in this paper (‘`ilc`’), an experimental variant which uses the higher-level data representation for algebraic types discussed in Section 3.8.3 (‘`il`’), and three existing back-ends (‘`asm_fast`’, ‘`asm_fast.gc`’, and ‘`hlc.gc`’)[4] For the `il` and `ilc` back-ends, the ‘`--verifiable-code`’ option was *not* enabled.

In addition, for two of the benchmarks we also measured the performance of equivalent programs written in C, compiled with MSVC (`cl`) or GNU C (`gcc -O3 -fomit-frame-pointer`), and in *C♯* , compiled with the Microsoft *C♯* compiler (`csc /o`).

We ran three different benchmarks, ‘`true`’, ‘`tak`’, and ‘`queens`’; the results are shown in table 4. All figures are execution times, measured in seconds.

Start-up time. To measure the impact on start-up times of initializing the Mercury RTTI data structures, we measured the time taken to execute a

[4] `asm_fast` is the original back-end of the Mercury compiler; it compiles to native code via low-level GNU C. It has no garbage collector, but heap space is recovered automatically on backtracking. `asm_fast.gc` is same as `asm_fast.gc`, except that it uses the Boehm (et al) conservative garbage collector [4]. `hlc.gc` is the MLDS-based high-level C back-end of the Mercury compiler; it compiles to native code via standard C. It too uses the Boehm collector.

trivial "do-nothing" Mercury program, `true`. When compiled to the .NET CLR, this program took about 1.8 seconds, compared with about 0.07 seconds for the same program compiled via C to native code, and about 0.16 seconds for a do-nothing $C\sharp$ program.

Recursion. `tak` is an artificial benchmark, originally written in Lisp; it is heavily recursive and does lots of simple integer arithmetic. We chose this benchmark because it was the only benchmark in our standard set that didn't use discriminated union types. To reduce the effects of start-up time, we wrote a test harness that ran the test 10000 times.

All versions ran at pretty similar speeds, with $C\sharp$ falling a little behind, probably because it was the only one that failed to perform tail call elimination. This shows that the CLR can match native code for at least some benchmarks.

Backtracking and lists. `queens` finds a solution to the problem of placing 11 queens on an 11x11 chess-board without any queen attacking any other queen; it makes heavy use of backtracking, lists, and heap allocation.

On this benchmark our CLR back-end does substantially worse than the other Mercury implementations; even when using a higher-level data representation rather than arrays of `System.Object`, it is a factor of 5 worse than the MLDS-based C back-end, and a factor of 10 worse than the best Mercury implementation, which recovers heap storage cheaply on backtracking, rather than using garbage collection.

A large part of this is no doubt due to the immature nature of our current implementation; for example, our compiler emits many unnecessary '`castclass`' instructions, which could easily be eliminated by common subexpression elimination. We emphasize that these benchmark results are preliminary.[5]

Overall, the benchmarks show widely varying results. No firm conclusions should be drawn about the overall performance competitiveness of compiling to the CLR from the benchmark results at this stage. However, the cost of initializing static data at start-up is definitely problematic for us.

5 Conclusions & Further work

We have implemented a compiler for Mercury that generates .NET CLR CIL code and which handles all of the standard Mercury language features.

By generating code for the CLR, many aspects of our language implementation have been simplified, because we have been able to make direct use of the high-level facilities that it offers, such as garbage collection and exception handling. But more importantly, having a compiler for the CLR is the first step towards achieving a much greater degree of interoperation between Mer-

[5] In fact, for the `il` version we had to hand-edit the generated CIL code slightly to make it work.

cury and a variety of other different languages that target the .NET CLR. Programmers can benefit from this interoperability with increased code reuse, e.g. by making use of the wide variety of existing components available on this platform.

We have also identified a number of areas in which the .NET CLR or future VMs could be improved to better support Mercury and other languages:

- support for static initialization of static data objects
- support for returning multiple values (instead of just one)
- more "first-class" support of by-ref and `refany` types
- allowing by-refs parameters for tail-calls in verifiable code
- verifiable function pointers
- verifier-enforced '`out`' mode parameters
- support for parametric polymorphism
- support for type synonyms as abstract data types

The work described in this paper is just a *first* step towards achieving easy language interoperability between Mercury and other languages on the CLR; it provides a base upon which we can then build specific compiler and/or language support for interoperating with other languages.

Future work includes tighter integration of the Mercury's type system and the CLR's type system, finding suitable interfaces for language feature mismatches, improving the efficiency of the code our compiler generates, and overcoming the remaining issues of verifiability. We would also like to do more detailed performance measurements.

Acknowledgements

Thanks to Zoltan Somogyi, Peter Schachte, Kevin Glynn, David Jeffery and the anonymous referees for reviewing earlier drafts of this paper.

A paragraph or two in the background material describing Mercury (Section 2.2) came from a paper by Thomas Conway and Zoltan Somogyi [5].

This work has been financially supported by grants from Microsoft to Zoltan Somogyi at the University of Melbourne, and by Mission Critical. Thanks to Microsoft, Mission Critical, and Zoltan Somogyi for their support, both financial and otherwise.

References

[1] R. Becket. Mercury tutorial, 1999. Available from `<http://www.cs.mu.oz.au/research/mercury/tutorial>`.

[2] N. Benton and A. Kennedy. Interlanguage working without tears: Blending SML with Java. In *International Conference on Functional Programming*, pages 126–137, 1999.

[3] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.

[4] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[5] T. Conway and Z. Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical Report 2001/24, Department of Computer Science, The University of Melbourne, Melbourne, Australia, July 2001.

[6] T. Dowd, Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery. Run time type information in Mercury. In *Proceedings of the 1999 International Conference on the Principles and Practice of Declarative Programming*, pages 224–243, Paris, France, September 1999.

[7] F. Henderson. Compiling Mercury to high-level C code. Submitted for publication; available from the author on request, June 2001.

[8] F. Henderson, T. Conway, and Z. Somogyi. Compiling logic programs to C using GNU C as a portable assembler. In *ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming*, pages 1–15, Portland, Oregon, 1995.

[9] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, and C. Speirs. The Mercury language reference manual. Available from `<http://www.cs.mu.oz.au/mercury/>`, 1995–2001.

[10] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.

[11] Microsoft. Microsoft .NET. `<http://www.microsoft.com/net/>`.

[12] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 1997.

[13] P. Tarau. Intelligent mobile agent programming at the intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.