



# Resource-bounded functional programming on the JVM and .NET

Stephen Gilmore

Mobile Resource Guarantees Project  
Laboratory for Foundations of Computer Science  
The University of Edinburgh

28th March 2002

<http://www.dcs.ed.ac.uk/home/stg/MRG/comparison>

---

---

# Comparing the JVM and .NET

- The Java Virtual Machine is an object-oriented execution environment for any language ***so long as it's Java.***
- The .NET platform is an object-oriented execution environment for any language ***so long as it isn't Java.***
- The .NET platform emphasises language inter-operability. Jim Miller, one of the architects of .NET said:

*I only want to do two, simple things. And I've wanted to do them for over thirty years:*

- 1. Write programs in the language I like, but use libraries written by other (less enlightened) people in other languages.*
  - 2. Write libraries in the language I like, but have them used by other (less enlightened) people from other languages.*
-

---

# Java Byte Code and MSIL

- **Java byte code** (or JVMML) is the low-level language of the JVM.
- **MSIL** (or CIL or IL) is the low-level language of the .NET Common Language Runtime (CLR).
- Superficially, the two languages look very similar.

JVML:	<code>iload_1</code>	MSIL:	<code>ldloc.1</code>
	<code>iload_2</code>		<code>ldloc.2</code>
	<code>iadd</code>		<code>add</code>
	<code>istore_3</code>		<code>stloc.3</code>

- One difference is that MSIL is designed only for JIT compilation. The generic **add** instruction would require an interpreter to track the data type of the top of stack element, which would be prohibitively expensive [Gou99].
-

---

# Type safety in the JVM and the CLR

- The JVM is intended to provide a type-safe execution environment where all Java byte code is “verified”, (it cannot forge pointers, cannot underflow the stack, . . . ). Any non-type-safe operations are regarded as errors.
  - The CLR is intended to provide a faithful execution environment for non-type-safe languages such as C (and Pascal, and others). Non-type-safe operations are regarded as inevitable.
  - As a multi-language platform, the CLR supports unsafe C-style pointers as well as managed references such as Visual Basic *byref* parameters.
  - As another example of this, the CLR provides variants on arithmetic instructions: one for languages in which overflow is treated as an *exception* (e.g. Standard ML and, I think, Pascal) and one for languages with *wrap around* (e.g. Java and C).
-

---

## Value types in the CLR

- The CLR supports non-object *value types*. These are stack-allocated sequences of named fields similar to *structs* in C or *records* in Standard ML and Pascal.

```
.class value Point {  
    .field public int x  
    .field public int y  
}
```

- The CLR supports C-style *union types* (or variant records in Pascal).

```
.class value explicit FloatOrInt {  
    .field [0] public float32 f  
    .field [0] public int32 n  
}
```

---

---

# Higher-order languages on .NET

- Functional languages include the lazy functional scripting language **Mondrian** [SPM02] which can be embedded in ASP.

```
// fibList : List<Integer>;
fibList =
    let fibHelper = a -> b ->
        a :: (fibHelper b (a+b));
    in fibHelper 1 1;
```

- Declarative languages include **P#** [Coo02] and **Mercury** [DHR01].

```
:- pred length(list(T), int).
:- mode length(in, out) is det.
length(L, N) :-
    ( L = [], N = 0
    ; L = [_Hd | T1], length(T1, N0), N = N0 + 1 ).
```

---

# Implementing functional languages

In implementing a functional language one of the challenges is that recursive function calls do not operate in constant space, whereas while loops do. There are three important kinds of function call.

```
fun fac 0 = 1                                not tail
  | fac n = n * fac (n - 1);                 recursive
```

```
fun fac (0, a) = a                           recursive
  | fac (n, a) = fac (n - 1, n * a);         tail call
```

```
fun fac (0, a) = a
  | fac (n, a) = fac2 (n - 1, n * a)         general
and fac2 (0, a) = a                          tail calls
  | fac2 (n, a) = fac (n - 1, n * a);
```

Non-tail recursive functions can be transformed into general tail recursive functions by ***continuation passing***.

---

## Tail call elimination

- The .NET CLR provides a **tail call** instruction. The following MSIL method (from [MM01]) will loop forever instead of overflowing the stack.

```
.method public static void Bottom() {  
    .maxstack 8  
    tail. call void Bottom(); ret  
}
```

- “*If the call is from untrusted code to trusted code the frame cannot be fully discarded for security reasons.*” [MM01]
  - Some Java Virtual Machines optimize *recursive* tails calls. (The IBM and Microsoft SDK do, but SUN’s JDK does not [SO01]). None of the JVMs optimize *general* tail calls.
    - Implementors claim that tail-call optimisations could cause problems for Java’s stack-walking security mechanism.
-

---

## Does tail call elimination matter?

When compiled with **MLj 0.1** (which does not perform tail call optimisations), the PEPA Compiler fails with a stack overflow although the same code compiled with another ML compiler completes successfully.

```
[tarff]stg: java -cp pepacompiler.zip pepacompiler
PEPA to PRISM compiler [version 0.021.5, 25-1-2002]
Filename: amani.pepa
Translating the model
Exception in thread "main" java.lang.StackOverflowError
    at G.ae(Unknown Source)
    at G.ae(Unknown Source)
    at G.ae(Unknown Source)
    ...
```

(“at G.ae(Unknown Source)” repeated 1024 times)

---

---

## Compiling tail calls

- A “brute force” method of removing tail calls is to put the entire program into a single function and simulate function calls by direct jumps or switch statements. A *whole-program* compiler such as **MLton** can do this, but not an incremental compiler.
  - This technique will not work on the JVM because method bodies cannot be more than 64Kb. However, the .NET CLR has no such restriction, so it can work there. (Godfrey Achola’s port of MLton to C# works in this way.)
  - Otherwise, one can use a *trampoline* [TAL90].

*“A trampoline is an outer function which repeatedly calls an inner function. Each time the inner function wishes to tail call another function, it does not call it directly but simply returns its identity (e.g. as a closure) to the trampoline, which then does the call itself.”* [SO02]
-

## Parameter passing by reference

The wish to be able to call other languages (and be called by them) means that compiled representations should have simple types. The following SML function could be compiled to MSIL as shown.

```

fun Swap                                .method static void Swap
  (xa: int ref,                          (int32& xa,
    ya: int ref)                          int32& ya) {
=
  .maxstack 2
let
  .locals (int32 z)
  val z = !xa                             ldarg xa; ldind.i4; stloc z
in
  ldarg xa; ldarg ya;
  xa := !ya;                               ldind.i4; stind.i4
  ya := z                                  ldarg ya; ldloc z; stind.i4
end;                                       ret }

```

Java calls by value so the JVM supports only one mode of parameter passing. The experience with the **Gardens Point Component Pascal** compiler shows that it is not trivial to implement other modes for the JVM [Gou00].

---

## Extensions to MSIL

- There is an extension of the MSIL bytecode called **ILX**, due to Don Syme [Sym01]. The purpose of this extension is to provide a better target for functional language compiler writers.
  - ILX extends MSIL with
    - first-class functions, closures and thunks;
    - parametric polymorphism;
    - discriminated unions;
    - first-class type functions.
  - An assembler translates these extensions into either regular or polymorphic MSIL instructions.
  - The translation is efficient and provides compiled representations with natural types **but** it uses an *unverifiable* module which implements closures using C-style function pointers.
-

---

# Higher-order languages and JVM

- Standard ML of New Jersey has recently been extended to parse and compile Java byte code class files via an extension of its strongly-typed intermediate language now called **JFlint** [LST02].
- **SML/JFlint** compiles Java byte code to run on the SML/NJ runtime system<sup>1</sup>. SML/JFlint is a *static* Java compiler with no dynamic class loading, reflection or native methods coded in C.
- The Java byte code is first compiled into a high-level, explicitly-typed, functional intermediate language called  **$\lambda$ JVM** which is then compiled to JFlint and then to **MLRISC** [Geo97].
- $\lambda$ JVM is described in [LTS01] as “a simply-typed lambda calculus expressed in A-normal form<sup>2</sup> and extended with the types and primitive instructions of the Java virtual machine”.

---

<sup>1</sup>... whereas MLj compiles SML source code to run on the Java runtime system.

<sup>2</sup>... functions and primitives are applied to values only.

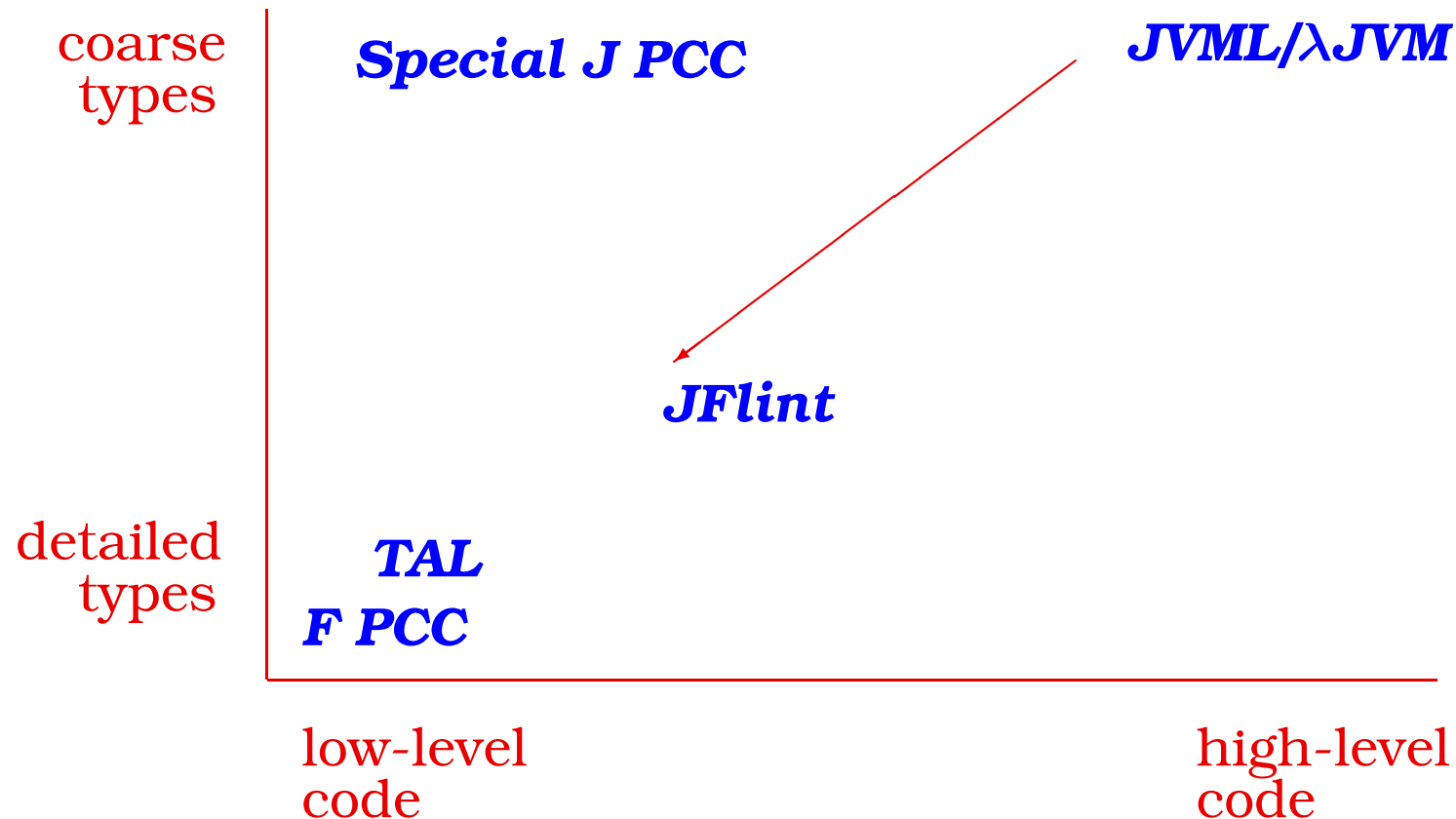
---

## SML/JFlint in operation

```
Standard ML of New Jersey v110.30 [JFLINT 1.2]
- Java.classPath := ["/home/league/r/java/tests"];
val it = () : unit
- val main = Java.run "Hello";
[parsing Hello]
[parsing java/lang/Object]
[compiling java/lang/Object]
[compiling Hello]
[initializing java/lang/Object]
[initializing Hello]
val main = fn : string list -> unit
- main ["Duke"];
Hello, Duke
val it = () : unit
- main [];
uncaught exception ArrayIndexOutOfBoundsException
  raised at: Hello.main([Ljava/lang/String;)V
```

# Language road map

In creating SML/JFlint the authors discovered some errors in the **Special J** proof-carrying code compiler [CLN+00]. This led them to suggest the following road map of typed intermediate languages(!).



---

## The authors' comments on related work

**Compared to ILX:** "... [ILX] added no fewer than 6 new types and 12 new instructions (bringing the total number of **call** instructions to 5) and it still does not support ML's higher-order modules or Haskell's constructor classes."

**Compared to MLj:** "JVML is less appropriate as an intermediate format for functional languages because it does not model their type systems well. Polymorphic code must either be duplicated or casts must be inserted. JFlint, on the other hand, completely models the type system of SML."

**Comparing the JVM and MSIL:** "Either favor one language and make everyone else conform (JVM) or incorporate the union of all the requested features (CIL, ILX)."

---

---

## Conclusions, omissions, etc

- The .NET platform goes further towards supporting other languages than the JVM, but at the cost of including undesirable features as well as desirable ones.
  - The recursive programming style, functional value types and stack allocation provide our major differences from the OO-programming style.
  - We had no discussion of *boxing* and *unboxing* (the Common Language instruction set supports this) and polymorphism (*generics* [KS01] are an extension to the Common Language instruction set).
-