Technical Overview of the Common Language Runtime (or why the JVM is not my favorite execution environment)

Erik Meijer and Jim Miller

{emeijer,jsmiller}@microsoft.com

Abstract

In the last few years many language researchers have moved to the JVM as the delivery vehicle for their languages. Although the JVM is a great target for the Java^{TM} programming language, it is not necessarily a good platform for other languages, especially languages that require semantic features that do not appear in Java^{TM}. In this paper we compare the JVM with the new Microsoft .NET Common Language Infrastructure (CLI), which has been designed from the ground up to be a multi-language platform.

1 Introduction

The ideas of virtual machines, intermediate languages and language independent execution platforms have fascinated language researchers for a long time. Well known examples include UNCOL [6], UCSD P-code [22], ANDF [20], AS-400 [24], hardware emulators such as VMWare, Transmeta Crusoe^{TM} [28], binary translation [25], the JVM [19], and most recently Microsoft's *Common Language Infrastructure* (CLI) [2].

There are several reasons why people are looking at alternative implementation paths for native compilers:

Portability By using an intermediate language, you need only $n+m$ translators instead of $n*m$ translators, to implement n languages on m platforms.

Compactness Intermediate code is often much more compact than the original source. This was an important property back in the days when memory was a limited resource, and has recently regained importance in the context of dynamically downloaded code.

Efficiency By delaying the commitment to a specific native platform as much as possible, you can make optimal use of the knowledge of the underlying machine, or even adapt to the dynamic behavior of the program.

Security High-level intermediate code is more amenable to deployment and runtime enforcement of security and typing constraints than low level binaries.

Interoperability By sharing a common type system and high-level execution environment (that provides services such as a common garbage collected heap, threading, security, etc), interoperability between different languages becomes easier than binary interoperability. Easy interoperability is a prerequisite for multi-language library design and software component reuse.

Flexibility Combining high level intermediate code with metadata enables the construction of (typesafe) metaprogramming concepts such as reflection, dynamic code generation, serialization, type browsing etc.

Attracted by the high-level runtime support and the wide availability of the JVM, and the rich set of libraries on the Java^{TM} platform, quite a number of language implementers have recently turned to the JVM as the execution environment for their language [27,7].

The JVM is a great target for Java^{TM}, but even though the JVM designers hope to attract implementers of other languages [19, Chapter 1.2], we will argue that the JVM is essentially a suboptimal multi-language platform.

For a start, the JVM provides no way of encoding type-unsafe features of typical programming languages, such as pointers, immediate descriptors (tagged pointers), and unsafe type conversions. Furthermore, in many cases the JVM lacks the primitives to implement language features that are not found in Java^{TM}, but are present in other languages. Examples of such features include unboxed structures and unions (records and variant records), reference parameters, varargs, multiple return values, function pointers, overflow sensitive arithmetic, lexical closures, tail calls, fully dynamic dispatch, generics, structural type equivalence etc [17,18,13,9,11,10,23].

The CLI has been designed from the ground up as a target for multiple languages, and explicitly addresses many of the issues mentioned above that are needed to efficiently compile a wide variety of languages. To ensure this, from early on in the development process of the CLI, Microsoft has worked closely with a large number of language implementers (both commercial and academic, for an up to date list see `www.gotdotnet.com`). For instance, the tail call instruction was added as a direct result of feedback from language researchers; tail calls are a necessary condition for efficiency in many declarative languages that use recursion as their sole way of expressing repetition.

It would be unfair to state that the CLI as it is now, is already the *perfect* multi-language platform. It currently has good support for imperative (COBOL, C, Pascal, Fortran) and statically typed OO languages (such as $C^\sharp$, Eiffel, Oberon, Component Pascal). Microsoft continues to work with language implementers and researchers to improve support for languages in non-standard paradigms [16].

In the remainder of this paper, we give a quick overview of the architecture, instruction set and type system of the CLI and point out specific points where we think the CLI is a better multi-language execution environment than the JVM.

2 Architecture of the Common Language Infrastructure (CLI)

The CLI manages multiple concurrent threads of control (which are not necessarily native OS threads). A thread can be viewed as a singly linked list of *stack frames*[12,3], where a frame is created and linked back to the current frame by a method call instruction, and removed when the method call completes (either by a normal return, a tailcall, or by an exception).

An instruction pointer (IP) which points to the next CLI instruction to be executed by the CLI in the present method.

An evaluation stack which contains intermediate values of the computation performed by the executing method (the *operand stack* in JVM terminology).

A (zero-based) array of local variables A local variable may hold any data type. However, a particular variable must be used in a type-consistent way (in the JVM, a local variable can contain an integer at one point in time and a float at another).

A (zero-based) array of incoming arguments Unlike the JVM the argument array and the local variable array are not the same.

A methodInfohandle which contains information about the method, such as its signature, the types of its local variables, and data about its exception handlers.

A local memory pool The CLI includes instructions for dynamic allocation of objects from the local memory pool (e.g. [3, Chapter 7.3, page 408].

A return state handle which is used to restore the method state on return from the current method. This corresponds to what in conventional compiler terminology would be the *dynamic link*.

A security descriptor which is used by the CLI security system to record security overrides (assert, permit-only, and deny). This descriptor is not directly accessible to managed code. Although extremely important and interesting, the security mechanism of the CLI is outside the scope of this paper.

In contrast to the JVM where all storage locations (local variables, stack slots, arguments) are 4 bytes wide, storage locations in the CLI are polymorphic, in the sense that they might be 4 bytes (such as a 32 bit integer) or hundreds of bytes (such as a user-defined value type), but their type is fixed for lifetime of the frame.

3 Assemblies

Every execution environment has a notion of "software component" [26]. An *assembly* is a set of files (modules) containing MSIL code and metadata, that serves as the primary unit of a software component in the CLI. Security, versioning, type resolution, processes (application domains) all work on a per assembly basis. In JVM terms an assembly could roughly be compared to a JAR file.

An assembly *manifest* describes information about the assembly itself, such as its version, which files make up the assembly, which types are exported from this assembly, and optionally a digital signature and public key of the manifest itself. Here is an example manifest for an assembly using ILASM syntax [2]:

```
.assembly HelloWorld {}

.assembly extern mscorlib {
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 1:0:2411:0
}
```

Inside an assembly or module we can define *reference* types such as classes, interfaces, arrays, delegates) (see section 7) and *value types* such as structs, enums (see section 6), and nested types. In contrast to the JVM, the CLI allows top-level methods and fields. All these declarations are included in the assembly's *metadata*. A unique feature of the CLI is that it's metadata is user extensible via the notion of custom attributes.

4 Type System

In this section we give an informal overview of the CLI type system, a more formal introduction is given by Gordon and Syme [8].

In addition to user defined types (section 6 and section 7), the CLI supports the following set of *primitive types*:

- `object`, shorthand for `System.Object`, `string`, shorthand for `System.String`, `void`, void return type.
- `bool`, 8-bit 2's complement signed value, `char`, 16-bit Unicode character.
- `int8`, `unsigned int8`, `int16`, `unsigned int16`, `int32`, `unsigned int32`, `int64`, `unsigned int64`, unsigned and 2's complement signed integers of respective width; `native int`, `unsigned native int`, machine dependent unsigned and 2's complement signed value.
- `float32`, `float64`, IEEE-754 floating point value of respective width; `native float`, machine dependent floating point number (not user visible).
- `typed reference`, an opaque descriptor of a pair of a pointer and a type, used for type safe varargs.

Primitive types can be combined into composite types using the following set of *type constructors*:

- `valuetype` *`typeref`*, `class` *`typeref`*, reference to value or reference type.
- *`type`* `pinned`, prevents the object at which local variable points from being moved by GC. This is outside the scope of this paper.
- *`type`*`[`*`bounds`*`]`, (multi-dimensional) array. This is outside the scope of this paper, suffice to note that in constrast to the JVM, the CLI *does* support true multi-dimensional arrays.
- `method` *`callConv type`*`*(`*`parameters`*`)`, function pointer. This is outside the scope of this paper.
- *`type`*`&`, managed pointer.
- *`type`*`*`, transient pointer (not user declarable).

The natural-size, or *generic*, types (primitive types `native int`, `unsigned native int`, `object`, and the two type constructors `&`, `*`) are a mechanism in the CLI for deferring the choice of a value's size. The CLI maps each to the natural size for a specific processor at JIT- or run-time. For example, a native int would map to `int32` on a Pentium processor, but to `int64` on an IA64 processor.

The `object` type represents an object reference that is managed by the CLI. A *managed pointer* `&` is similar to the `object` type, but points to the interior of an object. Managed pointers are not interchangeable with object references. *Transient pointers* `*` are intermediate between managed and unmanaged pointers. When a transient pointer is passed as an argument, returned as a value, or stored into a user-visible location it is converted either to a managed pointer or an unmanaged pointer depending on the type specified for the destination.

Natural sized types offer a significant advantage over the JVM which prematurely commits all storage locations to be 32 bits wide. This implies for example that values of type `long` or `double` occupy two locations, which makes things unnecessarily hard for compiler writers.

A more important weakness of the JVM as a target for multiple language is the fact that its type system lumps together all pointers into one `reference` type, closing the door for languages or compilers that do need a more fine-grained level of detail. We will expand on the usefulness of the CLI pointer types in more detail in section 9.

5 Base Instruction set

The CLI has about 220 instructions, so obviously we do not have space to cover all of them in this paper, instead we will highlight a few representative instructions from each group below [1].

When comparing to JVM instructions, you will notice that unlike the JVM where most instructions have the types of their arguments hard-coded in the instruction (which makes it easier to *interpret* JVM byte code, but puts a burden on every compiler that generates JVM byte codes), the CLI instruction set is much more polymorphic and usually only requires explicit type information for the result of an instruction (which makes it easier for compilers to generate MSIL code, but requires more work from the JIT).

5.1 Constants, arguments, local variables, and pointers

The CLI provides a number of instructions for transferring values to and from the evaluation stack. Instructions that push values on the evaluation stack are called "loads", and instructions that pop elements from the stack into local variables are called "stores".

[1] Many of the CLI instruction also have short forms, that allow more compact representation in certain special cases. We will not discuss these variants here

The simplest load instruction is `ldc.`*t* *v*, that pushes the value *v* of type *T* [2] on the evaluation stack. The `ldnull` pushes a null reference (of type `object`) on the stack.

The `ldarg` *n* instruction pushes the contents of the *n*-th argument on the evaluation stack. The `ldarga` *n* instruction pushes the *address* (as a transient pointer of type *T*`*`) of the *n* argument on the evaluation stack. The `starg` *n* instruction pops a value from the stack and stores it in the *n*-th argument. In each case, the JIT knows the type of the value from the signature of the method.

The `ldloc` *n* instruction pushes the contents of the *n*-th local variable onto the evaluation stack, and `ldloca` *n* pushes the address of the *n*-th local variable on the evaluation stack. The `stloc` *n* instruction pops a value from the stack and stores it in the *n*-th argument. Again, the JIT can figure out the types of these values from the context.

The `ldind.`*t* instruction expects an address (which can be a raw pointer, a managed, or a transient pointer) on the stack, dereferences that pointer and puts the value on the stack. The `stind.`*t* *v* instruction stores a value *v* of type *T* at address found at the top of the stack. In both cases, the type *t* is needed because the JIT cannot always infer what the type of the resulting value is.

The other load and store instructions include `ldfld`, `ldsfld`, `stfld`, `stsfld`, and `ldflda` and `ldsflda` to manipulate instance and static fields, and a similar family of instructions for arrays.

5.1.1 Example: reference arguments

The ability to load the address of local variables, and to dereference pointers to indirectly get the value they point at allows compiler writers to efficiently implement languages that support passing arguments by reference. For example, here is the MSIL version of the `Swap` function that swaps the values of two variables:

```
.method static void Swap(int32& xa, int32& ya) {
  .maxstack  2
  .locals (int32 z)
  ldarg xa; ldind.i4; stloc z
  ldarg xa; ldarg ya; ldind.i4; stind.i4
  ldarg ya; ldloc z; stind.i4
  ret
```

[2] Here *T* ∈ {`int32`, `int64`, `float32`, `float64`} and *t* is the short form of *T*. The short form of types is used in all instructions that have a type index.

```
}
```

To call this function (see section 8), we just pass the addresses of the local variables as arguments to function `Swap`:

```
.locals (int32 x, int32 y)

// initialize x and y

ldloca   x
ldloca   y
call     void Swap(int32&, int32&)
```

In the JVM there is a separate load (and store) instruction for each type, i.e. `iload_`*n* pushes the integer content of the *n*-th local variable on the stack, and similarly for `aload_`*n* (reference), `dload_`*n* (double, so it will moved as two 32 bit values), `fload_`*n* (float), and `lload_`*n* (long, again, moves two items will be moved).

The JVM does not allow compilers to take the address of local variables, hence it is impossible to implement byref arguments directly. Instead compiler writers have to resort to tricks such as passing one-element arrays, or by introducing explicit box classes (the JVM does not support boxing and unboxing either). Gough [11] gives a detailed overview of the intricate design space of implementing reference arguments on the JVM.

5.2 Arithmetic

The `add` instruction adds the two topmost values on the stack together (and similarly for other arithmetic instructions). Overflow is not normally detected for integral operations unless you specify `.ovf` (signed) or `ovf.un` (unsigned); floating-point overflow returns $+\infty$ or $-\infty$.

The JVM *never* indicates overflow during operations on integer data types, which means that the time penalty may be significant for procedures which perform intensive arithmetic in languages (such as Ada95 [1] or SML [21]) that require overflow detection. A minor issue in this context, is that there is a separate `add` instruction for each type (and similar for other arithmetic instructions), just as is the case for load and store.

5.3 Simple control flow

The CLI supports the usual variety of (conditional) branch instructions (such as `br`, `beq`, `bge` etc.). There is no analogy of the JVM "jump subroutine"

instruction. Also the CLI does not limit the length of branches to 64K as the JVM does (which might not be a big deal for humans programming in Java, but it is a real problem for compilers generating JVM byte code).

6 Value Types

A *value type* is similar to a struct in C or record in Pascal, i.e. a sequence of named fields of various types. In contrast to reference types, which are always allocated on the GC heap, value types are allocated "in place". In the CLI, value types can also contain (static, virtual, or instance) methods [2], the details of which are outside the scope of this paper.

6.1 Structures

Here is the definition of a simple `Point` structure that contains two fields `x` and `y` (which the CLI may store in any order):

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

6.2 Unions

The CLI also supports sequential and explicit layout control of fields. The latter is needed to implement C-style *union types* (or variant records in Pascal), a structure where the fields may overlap. For example the following value class defines a union that may hold either a float or an int:

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

6.3 Enums

Besides structures, there is another kind of value type, *enumerations*, which correspond to C-style enums. Enumerations provide a type safe way to associate names with integer values. For example the following enum defines a new value type `Shape` with two constants `RECTANGLE` and `CIRCLE`:

```
.class enum Shape {
```

```
    .field public static valuetype Shape RECTANGLE = int32(0)
    .field public static valuetype Shape CIRCLE    = int32(1)
  }
```

The CLI also allows you to specify enum details such as the internal storage type or indicating that the enumeration is a collection of bits, for more details see [2].

6.4 Initializing valuetypes

Except for boxing and the `.locals` directive, the CLI does not have special mechanisms or instructions to explicitly allocate memory for a valuetype. The `initobj` *T* instruction expects the address of a valuetype *T* on the stack, and initializes all the fields of the valuetype to either `null` or a 0 of the appropriate primitive type (this is a nice example of a *polytypic* instruction). For example to initialize the example `Point` struct that we introduced in section 6.1, we would load the address of the local variable `p` of type `Point` on the stack and call `initobj Point`:

```
.locals (valuetype Point p)

ldloca   p
initobj  Point
```

It should be obvious that having value types is essential for compiling Pascal or C-like languages that have enums, record and union types. Compiling such languages to the JVM is inefficient to start with, as you need to represent enums and structs by classes and unions by class hierarchies [4, Chapter5]. A much more serious consequence is that it is impossible to support the full semantics of such languages, as it is impossible to implement the common (type unsafe) trick where you store a a float in an `FloatOrInt` union type, and read it as an int:

```
.locals (valuetype FloatOrInt fi, int32 n)

 // fi.f = 3.14
 ldloca     fi
 ldc.r4     3.14
 stfld      float32 FloatOrInt::f
 // n = fi.n
 ldloca     fi
 ldfld      int32 FloatOrInt::n
```

7 Reference types

The CLI supports types such as classes, interfaces, arrays, delegates. Because of lack of space, we will restrict our attention to classes. Classes can contain methods and fields; but yet again, to support as many languages as possible, besides virtual and static methods (as in Eiffel, and Java^{TM}), the CLI also support instance methods (as in C++).

For example, here are two classes `Foo` and `Bar` that both define an instance method `f`, and a virtual method `g`:

```
.class public Foo {
    .method public virtual void f() { ... }
    .method public instance void g () { ... }
    .method public static void h () { ... }
    .method public specialname void  .ctor() { ... }
  }

.class public Bar extends Foo {
  .method public virtual void f() { ... }
  .method public instance void g () { ... }
  .method public static void h () { ... }
  .method public specialname void .ctor() { ... }
}
```

Constructors always are names `.ctor` and have to be marked as `specialname`.

7.1 Instantiating Reference types

The `newobj c` instruction allocates a new instance of the class associated with constructor `c` and initializes all the fields in the new instance. It then calls the constructor with the given arguments along with the newly created instance.

For example, we can create an instance `f` with static type `Foo` of our class `Foo`, and an instance `b` with static type `Foo` of our class `Bar` using the following instruction sequence:

```
.locals (class Foo f, class Foo b)

newobj void Foo::.ctor(); stloc f
newobj void Bar::.ctor(); stloc b
```

To create an instance of a class `c` in the JVM, you always have to use the sequence `new c; dup; invokespecial c.<init>()V` (and similarly for using a constructor that takes arguments) and the Java^{TM} verifier must do a com-

plex dataflow analysis to ensure that no object is used before it is properly initialized or that it is initialized more than once [19, Chapter 4.9.4]. It seems much simpler to avoid all the complexity to start with and just do allocation and initialization in a single instruction.

8 Invoking methods

The CLI has two call instructions for directly invoking methods and interfaces. A third call instruction `calli` allows indirect calls on a function pointer, but this is outside the scope of this paper.

The `call` *m* instruction is normally used to call a static method *m* (i.e. it is comparable to the `callstatic` instruction in the JVM). For example, to call method `Foo::h()`, we just write:

```
call void Foo::h()
```

It is legal to call a virtual or instance method using `call instance` (rather than `callvirt`); in which case method lookup is done statically, in other words, you will get an early bound call (i.e. the effect is comparable to a `invokespecial` on the JVM). Assuming that `bar` is a local variable that contains an instance of class `Bar`, the following call would actually execute method `Foo::f()`:

```
ldloc bar;
call instance void Foo::f()
```

The `instance` calling convention indicates that `Foo::f()` expects an additional "this" parameter.

The `callvirt` *m* instruction makes a late bound call to a virtual method *m*, in other words, the actual method that is invoked depends on the dynamic type of the "this" parameter (the JVM has two separate instructions, `invokevirtual` and `invokeinterface` for this purpose, which once again makes life harder for compiler writers). So in the example below, the method that will be invoked is `Bar::f()` since the this parameter passed to the call has static type `class Foo`, but dynamic type `class Bar`:

```
ldloc bar;
callvirt void Foo::f()
```

For instance methods, `callvirt` will still result in an early bound call.

8.1 Tailcalls

Some people find it hard to believe, but there are programming languages where recursion is the only way of expressing repetition (examples include Haskell, Scheme, Mercury). For these languages, it is essential that the underlying execution environment supports tailcalls. The `tail.` prefix instructs the JIT compiler to discard the caller's stack frame prior to making the call, which means that the following method will indeed loop forever instead of throwing a stack overflow exception:

```
.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.

Since the JVM does not support tailcalls, compiler writers are forced to use tricks like trampolines to artificially force the JVM to discard stack frames [5,15,14,18].

9 Interaction between value and reference types

If you have both valuetypes and reference types, programmers will want to use valuetypes in contexts where reference types are required (for instance to store a `Point` in a collection). The same problem occurs in dynamic languages like Scheme and statically typed polymorphic functional languages like Haskell and SML where polymorphic functions expect a uniform argument representation.

To support these scenarios, it is essential to have efficient support from the execution environment to move between the worlds of value- and reference types. Having to create an instance of a class every time you want to pass a valuetype as a reference type has too much performance overhead. Moreover, this would also force you to define a new class for every valuetype, or introduce many unnecessary casts.

The CLR provides built-in support for boxing and unboxing. A valuetype *`T`* can be turned into reference type `object` using the `box` *`T`* instruction, and back into a valuetype using the `unbox` *`T`* instruction.

10 Conclusions and future work

In the previous sections we have argued that the CLI is already strictly more powerful than the JVM as a multi-language platform. Microsoft Research and the .NET product group continue to work with language inplementors to improve support a wide variety of language paradigms.

We explictly solicit language implementors (including those who now target the JVM) to try to target the CLI and provide us with feedback on how we can make the CLI even better than it is today.

Acknowledgements

We would like to thank all *Project 7* participants, and Patrick Dussud, Jim Hogg, Clemens Szysperski, Don Syme, Andrew Kennedy, and Nick Benton, for many discussions on the topics discussed in this paper. Nicks's notes on his previous experiences with compiling SMLj to the JVM were especially helpful.

References

[1] Ada 95 Reference Manual, 1995. ANSI/ISO/IEC-8652:1995.

[2] CLI Partition II: Metadata. http://msdn.microsoft.com/net/ecma/, 2001. ECMA TG3.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1986.

[4] Joshua Bloch. *Effective Java Programming Language Guide*. Addison Wesley, 2001.

[5] Per Bothner. Kawa —Compiling Dynamic Languages to the Java VM. In *USENIX'98 Technical Conference*, 1998.

[6] M. E. Conway. Proposal for an UNCOL. *CACM*, 1(10):5–8, 1958.

[7] J. Engel. *Programming for the Java Virtual Machine*. Addison Wesley, 1999.

[8] Andrew Gordon and Don Syme. Typing a Multi-Language Intermediate Code. In *Proceedings POPL'01*, pages 248–260, 2001.

[9] J. Gough. Parameter Passing for the Java Virtual Machine. In *Proceedings of the Australasian Computer Science Conference*, 1998.

[10] J. Gough. Stacking them up: A Comparison of Virtual Machines. In *Proceedings ACSAC-2001*, 2001.

[11] J. Gough and D. Corney. Evaluating the Java Virtual Machine as a Target for Languages other than Java. In *Proceedings Joint Modula Languages Conference*, 2000.

[12] Dick Grune, Henri Bal, Ceriel Jacobs, and Koen Langendoen. *Modern Compiler Design*. Wiley, 2001.

[13] Jonathan C. Hardwick and Jay Sipelstein. Java as an Intermediate Language. Technical Report CMU-CS-96-161, Carnegie Mellon University, August 1996.

[14] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C–: a Portable Assembly Language that Supports Garbage Collection. In *International Conference on Principles and Practice of Declarative Programming*, 1999.

[15] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, 1978.

[16] Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings PLDI'01*, 2001.

[17] Andreas Krall and Jan Vitek. On Extending Java. In Hanspeter Mössenböck, editor, *Joint Modular Languages Conference (JMLC'97)*, pages 321–335, Linz, 1997. Springer.

[18] Christopher League, Zhong Shao, and Valery Trifonov. Representing Java Classes in a Typed Intermediate Language. In *International Conference on Functional Programming*, pages 183–196, 1999.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2e)*. Addison Wesley, 1999.

[20] S. Macrakis. From UNCOL to ANDF: Progress In standard Intermediate Languages. Technical report, Open Software Foundation Research Institute, 1993.

[21] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[22] P. A. Nelson. A Comparison of PASCAL Intermediate Languages. *ACM SIGPLAN Notices*, 14(8):208–213, 1979.

[23] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[24] David L. Schleicher and Roger L. Taylor. System Overview of the Application System/400. *IBM Systems Journal*, 38(2/3):398–413, 1999.

[25] R. Sites, A. Chernoff, M. Kirk, and M. Marks. Binary Translation. *CACM*, 36(2):69–81, 1993.

[26] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.

[27] Robert Tolksdorf. Programming Languages for the Java Virtual Machine. http://grunge.cs.tu-berlin.de/ tolk/vmlanguages.html.

[28] TRANSMETA. The Technology behind Crusoe Processors. http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf, 2000.