Java Interoperability in Managed X10

Mikio Takeuchi David Cunningham David Grove Vijay Saraswat

IBM Research - Tokyo IBM T. J. Watson Research Center

mtake@jp.ibm.com, dcunnin,groved,vsaraswa@us.ibm.com

Abstract

The ability to smoothly interoperate with other programming languages is an essential feature to reduce the barriers to adoption for new languages such as X10. Compiler-supported interoperability between Managed X10 and Java was initially previewed in X10 version 2.2.2 and is now fully supported in X10 version 2.3. In this paper we describe and motivate the Java interoperability features of Managed X10. For calling Java from X10, external linkage for Java code is explained. For calling X10 from Java, the current implementation of Java code generation is explained.

An unusual aspect of X10 is that, unlike most other JVM-hosted languages, X10 is also implemented via compilation to C++ (Native X10). The requirement to support multiple execution platforms results in unique challenges to the design of cross-language interoperability. In particular, we discovered that a single top exception type that covers all exception types from source and all target languages is needed as a native type of the source language for portable exception handling. This realization motivated both minor changes in the X10 language specification and an extensive redesign of the X10 core class library for X10 2.3.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, compilers

General Terms Languages, Design

Keywords X10, Java, multi-platform, interoperability, unified type system, code generation, exceptions, primitives, arrays, generics

1. Introduction

The X10 programming language [11] was originally designed as a "better Java" – Java [6] with PGAS [10] extensions for distributed computing and enhancements for true multi-dimensional arrays. X10 was implemented solely by compilation to Java and used the JVM as its only execution platform. Most existing Java programs were also valid X10 programs.

However, through major revisions of the language, X10 has been redesigned as a full-fledged programming language whose implementation targets multiple execution substrates. In addition to the original Managed X10, which continues to target the JVM, Native X10 targets the C++ [14] and CUDA (GPGPU) [2] execution platforms. X10 also introduced more features from mainstream and modern programming languages. The introduction of Scala [12]-like syntax in X10 1.7 meant that most existing Java code was no longer valid X10 code. The changes were not limited to the way in which X10 programs look like. The introduction of reified generics like C++ templates made the efficient execution of X10 programs on Java platform require novel implementation techniques that were described in [15, 16] (distributed GC is described in [8]), and complicated the interoperation with existing Java code.

The ability to interoperate with other programming languages is an essential feature for the new languages such as X10. Cross-language interoperability enables both the incremental adoption of X10 in existing applications and the usage of existing libraries and frameworks by newly developed X10 programs. There are two primary interoperability scenarios that are supported by X10: inline substitution of X10 constructs (types, methods, fields, and statements) with foreign code fragments and external linkage to foreign code.

The design of X10/Java interoperability was motivated and informed by our experience in developing M3R: an X10 Main Memory Map Reduce engine [13]. The core M3R engine is several thousand lines of X10 code which provides an X10 Map/Reduce API against which application programs can be written. This core engine (and programs that use it) can be compiled with either Managed X10 or Native X10 to enable flexibility in selecting the execution platform to match the desired performance characteristics and/or need to link with existing application libraries. On top of the core M3R engine, there is also a Hadoop API compatibility layer that allows unmodified Hadoop Map/Reduce jobs to be executed on the M3R engine. The Hadoop compatibility layer is written using a mix of X10 and Java code and heavily uses the Java interoperability capabilities of Managed X10. Our work with M3R/Hadoop drove the initial design of Java interoperability in X10 2.2.3 and the experience of "hardening" M3R into a production server motivated the redesign in X10 2.3 in the area of exception handling.

In the rest of this paper we describe the Java interoperability features of Managed X10. The description is based on X10 2.3.1 unless otherwise specified. Section 2 provides background information on the Managed X10 and Native X10. Section 3 explains a simple text-based macro mechanism for inline substitution with foreign code fragments, which is available in all X10 implementations. The next two sections describe Java interoperability in Managed X10: Section 4 describes external linkage to Java code and Section 5 shows linking Java to X10. After discussing how other JVM-based languages implement Java interoperability in Section 6, Section 7 offers our conclusions.

2. Review of Managed and Native X10

X10 is implemented via source-to-source compilation to another language, which is then compiled and executed using that language's existing tool chain. X10 2.3 supports two such implementation paths: Managed X10, which is X10 compiled to Java, and Native X10, which is X10 compiled to C++.

The overall architecture of the X10 compiler is depicted in Figure 1. This compiler is composed of two main parts: an AST-based front-end and optimizer that parses X10 source code and performs AST based program transformation; Native/Java backends that translate the X10 AST into C++/Java source code and invokes a post compilation process that either uses a C++ compiler to produce an executable binary or a Java compiler to produce bytecode.

X10'13, June 20, 2013, Seattle, Washington.

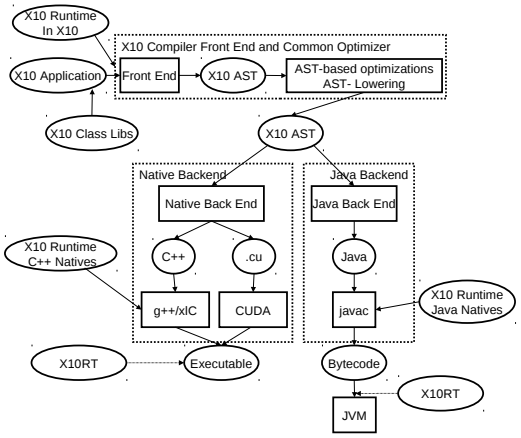

Figure 1. X10 Compiler Architecture

This final executable form can then either be executed directly or loaded and run by a JVM.

Additional details on the implementation and performance models of Managed and Native X10 can be found in [7] and in the online documentation at [17].

3. Inline Substitution with Foreign Code Fragments

This section describes a primitive mechanism for foreign language interoperability that provides inline substitution of X10 constructs (types, methods, fields, and statements) with foreign code fragments. This basic mechanism is supported by the X10 compiler for all X10 target platforms (Java, C++, CUDA).

What is provided is a simple text-based simple macro mechanism, which supports both positional and keyword (preferred) parameters, in forms of either `@NativeRep(lang,useType,defType, runType) type` or `@Native(lang,code) methodFieldStmt`. When they are compiled for the target language `lang`, the `type` is substituted with either `useType` or `defType`[1] and runtime type of `type` is substituted with `runType`. The `methodFieldStmt` is substituted with `code`[2].

The advantages of this mechanism are its simplicity and universality. The substitution is done by the front-end of X10 compiler, thus it is available in all X10 implementations and can be used for conditional programming for different target languages by attaching multiple `@NativeRep` or `@Native` annotations to the same X10 construct, as shown in Figure 2. Native X10 uses an additional set of similar annotations to indicate external header files that need `#include` directives in the generated file and to specify additional C++ source files or libraries that should be compiled and linked along with the generated C++ code.

The downside of these mechanisms is that there is no integration with X10 type system. The type system does not infer their types from foreign code fragments[3], nor type check them. X10 programmers need to not only specify their types explicitly, but also be careful to write correct code fragments in the target language to avoid post-compilation errors.

[1] `defType` is not used in Managed X10.

[2] In user programs, `@Native` annotations that are attached to non-final virtual methods are ignored.

[3] In current implementation, the type system does not infer their types even if they have X10-based implementation, too.

```
package x10.lang;
import x10.compiler.Native;
import x10.compiler.NativeRep;

@NativeRep("java", "java.lang.Object", null, "x10.rtt.Types.ANY")
@NativeRep("c++", "x10::lang::Any*", "x10::lang::Any", null)
public interface Any {
  @Native("java", "x10.rtt.Types.toString(#this)")
  @Native("c++", "x10aux::to_string(#this)")
  def toString():String;

  @Native("java", "x10.rtt.Types.typeName(#this)")
  @Native("c++", "x10aux::type_name(#this)")
  def typeName():String;

  @Native("java", "((java.lang.Object)(#this)).equals(#that)")
  @Native("c++", "x10aux::equals(#this,#that)")
  def equals(that:Any):Boolean;

  @Native("java", "x10.rtt.Types.hashCode(#this)")
  @Native("c++", "x10aux::hash_code(#this)")
  def hashCode():Int;
}
```

Figure 2. Inline substitution example (the definition of `x10.lang.Any`)

4. External Linkage to Java Code

Unlike Native X10, Managed X10 has a type-safe mechanism to interoperate with existing Java code. Because this mechanism is much easier to use than macro based ones, it is the recommended way for writing X10 applications that interoperate with existing Java code. Managed X10 does not pre-process existing Java code to make it accessible from X10. The generated Java code by Managed X10 compiler directly calls existing Java code as is. The key capability that enables this mechanism is support in the X10 compiler for processing Java class files and creating a representation in the X10 type system and abstract syntax trees for the entities contained in them. In this section, we describe how various Java entities are seen in X10.

Types Java classes are seen as X10 classes. Java enums are seen as X10 final classes. Java interfaces are seen as X10 interfaces.

In Managed X10, both at compile time and run time, there is no way to distinguish Java types from X10 types. Java types can be referred to with regular `import` statements, or their qualified name. The package `java.lang` is not auto-imported into X10. The resolver is enhanced to resolve types with X10 source files in the source path first, then resolve them with Java class files in the class path. Note that the resolver does not resolve types with Java source files, therefore Java source files must be compiled before compiling X10 source files that refer to the Java types. To refer to Java types listed in Tables 1 and 2, which include all Java primitive types, use the corresponding X10 type (e.g. `x10.lang.Int` (or in short, `Int`) instead of `int`). These X10 types are alias of Java types and the Java types are hidden from the X10.

Fields Fields of Java types are seen as fields of X10 types.

Since X10 does not have mutable static fields, there is no natural way to set values to Java static fields. To workaround such situation, Managed X10 has a utility class `x10.interop.Java` to support Java interoperability. By using its static methods `Java.setStaticField[T](name:String,value:Int)` etc., we can set values of any type to Java static fields, as shown in Figure 3.

Managed X10 does not change the static initialization semantics of Java types, which is per-class and at load time, therefore, it is subtly different than the per-field and lazy (at first access) initialization semantics of X10 static fields.

```
public class J {
  public static String s = "hello, ";
  public static void main(String[] args) {
    System.out.print(s);
    s = "world";
    System.out.println(s);
  }
}

import x10.interop.Java;
public class X {
  public static def main(Rail[String]):void {
    Console.OUT.print(J.s);
    Java.setStaticField[J]("s","world");
    Console.OUT.println(J.s);
  }
}

> javac J.java
> java J
hello, world
> x10c -cp . X.x10
> x10 X
hello, world
```

Figure 3. Java fields in X10

```
public class J {
  public static int sum(int... nums) {
    int sum = 0;
    for (int num : nums) sum += num;
    return sum;
  }
  public static void main(String[] args) {
    int sum = sum(1,2,3);
    System.out.println(sum);
  }
}

import x10.interop.Java;
public class X {
  public static def main(Rail[String]):void {
    //val sum = J.sum(1,2,3); // wrong
    val sum = J.sum(Java.convert([1,2,3])); // OK
    Console.OUT.println(sum);
  }
}

> javac J.java
> java J
6
> x10c -cp . X.x10
> x10 X
6
```

Figure 4. Java methods in X10

Methods Methods of Java types are seen as methods of X10 types.

X10 does not support variadic methods. To invoke Java method that takes variable number of arguments, X10 programmers need to create a Java array that includes all arguments and pass it to the method as shown in Figure 4.

Constructors Constructors of Java types are seen as constructors of X10 types.

Generic types Generic Java types are seen as their raw types (§4.8 in [6]). Raw type is a mechanism to handle generic Java types as non-generic types, where the type parameters are assumed as `java.lang.Object` or their upperbound if they have it. Java introduced generics and raw type at the same time to facilitate generic Java code interfacing with non-generic legacy Java code. Managed X10 uses this mechanism for a slightly different purpose. Java erases type parameters at compile time, whereas X10 preserves their values at run time. To manifest this semantic gap in generics, Managed X10 represents Java generic types as raw types and eliminates type parameters at source code level. For more detailed discussions, please refer to [15, 16].

Arrays X10 array type is a generic type and its representation is different from Java array types.

Managed X10 provides a special X10 type `Java.array[T]` to represent Java array type `T[]`. Note that for X10 types in Table 1, this type means the Java array type of their primary type. For example, `Java.array[Int]` means `int[]`. Managed X10 also provides conversion methods[4] between X10 array of rank 1 `Rail[T]`[5] and Java array (i.e. `Java.convert[T](Rail[T]):Java.array[T]` and `Java.convert[T](Java.array[T]):Rail[T]`), and construction methods for Java arrays (e.g. `Java.newArray[T](Int)`: `Java.array[T]`). Java array elements can be accessed with the same syntax as for X10 arrays, as shown in Figure 5.

Exceptions For multi-platform languages such as X10, it is important for cross-language interoperability to have a single top exception type *as a native type of source language* and cover all exceptions from the source and all the target languages with it. Figure 6 shows the type hierarchy in X10 2.2.2, when we previewed Java interoperability with Managed X10 for the first time. As you notice, there were two independent exception hierarchies topped by `x10.lang.Throwable` and `java.lang.Throwable`. Because the latter one is a Java type which is visible only in Managed X10, there was no way to implement portable X10 libraries that can be used in cross-language interoperability scenarios.

Before X10 2.3, there were two reasons for not being able to have a single top exception type. One reason was the presence of `x10.lang.Object` as a supertype of all reference types which include X10 exception types. Because `x.l.Object` was compiled to some type differ from `java.lang.Object`, which is the alias of `x10.lang.Any`, and all existing Java types do not have such X10 unique type as their supertype, it was not possible to map Java exception types to X10 exception types.

Another reason was the lack of checked exceptions. Unlike Java exception types, all X10 exception types were (and still are) unchecked exceptions, therefore X10 did not have checked exceptions nor `throws` clause. To handle Java checked exceptions in that situation, X10 compiler needs to generate code that catches checked exceptions and converts them to unchecked ones at each call site of Java method that may throw checked exceptions. It not only complicates the code generation but slows down the invocation of Java methods.

To solve these issues, we removed `x.l.Object` and introduced checked exceptions and `throws` clause in X10 2.3.

Figure 7 shows the type hierarchy of X10 2.3. As you notice, there is a single top exception type `x10.lang.CheckedThrowable`.

[4] These conversion methods do not copy backing arrays.

[5] Currently (in X10 2.3.1), `Rail[T]` is an alias of `Array[T](1)`. In X10 2.4, we will reimplement `Rail[T]` as a separate type that represents a dense, zero based, one dimensional X10 array and replace the most use of `Array[T](1)` with it for better performance.

```
import x10.interop.Java;
public class X {
  public static def main(Rail[String]):void {
    try {
      val a = Java.newArray[Int](2);
      a(0) = 0;
      a(1) = 1;
      a(2) = 2;
    } catch (e:ArrayIndexOutOfBoundsException) {
      Console.OUT.println(e);
    }
  }
}

> x10c -o X.jar X.x10
> x10 -cp X.jar X
x10.lang.ArrayIndexOutOfBoundsException: \
Array index out of range: 2
```

Figure 5. Java exceptions in X10

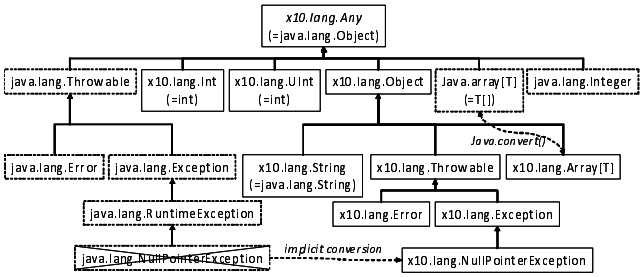

Figure 6. Type hierarchy in X10 2.2.2

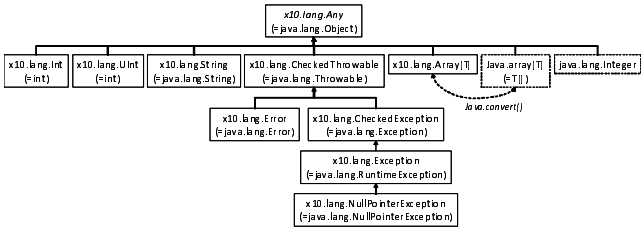

Figure 7. Type hierarchy in X10 2.3

By mapping it to `java.lang.Throwable` in Managed X10, we can now implement portable X10 libraries that handle any exceptions from both the source and all the target languages. In addition, the new X10 type hierarchy was designed so that there is a natural correspondence with the Java exception hierarchy. As shown in Table 2, many X10 exception types are alias of Java exception types. Note that Java exception types that are aliased can (and must) be caught (and thrown) using their X10 types, as shown in Figure 5.

5. Linking Java to X10

Managed X10 translates X10 programs to Java class files.

X10 does not provide a Java reflection-like mechanism to resolve X10 types, methods, and fields with their name at runtime, nor a code generation tool, such as `javah`, to generate stub code to access them from other languages. Java programmers, therefore, need to access X10 types, methods, and fields in the generated Java code directly as they access Java types, methods, and fields. To make it possible, Java programmers need to understand how X10 programs are translated to Java.

Some aspects of the X10 to Java translation scheme may change in future version of X10; therefore in this document only a stable subset of translation scheme will be explained. Although it is a subset, it has been extensively used by X10 core team and proved to be useful to develop Java Hadoop [4] interop layer for a Main-memory Map Reduce (M3R) engine [13] in X10.

In the following discussions, we deliberately ignore generic X10 types because the translation of generics is an area of active development and will undergo some changes in future versions of X10. For those who are interested in the implementation of generics in Managed X10, please consult [16]. We also do not cover function types, function values, and all non-static methods. Although slightly outdated, another paper [15], which describes translation scheme in X10 2.1.2, is still useful to understand the overview of Java code generation in Managed X10. We also do not cover how X10's parallel constructs such as `async`, `finish` etc. are translated to Java. In the generated Java code, types, fields, and methods have the same name as in X10 code unless otherwise specified.

Types X10 classes and structs are translated to Java classes. X10 interfaces are translated to Java interfaces.

Table 1 shows the list of special types that have two representations in the generated Java code. Primitives are their primary representations that are useful for good performance. Wrapper classes are used when the reference types are needed. Each wrapper class has two static methods `$box()` and `$unbox()` to convert its value from primary representation to wrapper class, and vice versa. Managed X10 compiler inserts their calls as needed, but Java programmers need to insert them manually for passing Java primitives to X10 method whose parameter type is `Any` or `T`[6]. As you notice, every unsigned type uses the same Java primitive as its corresponding signed type for its representation.

Table 2 shows the list of another kind of special types that are mapped (not translated) to Java types. As you notice, since the interface `Any` is mapped to the class `java.lang.Object` and it is hidden from X10, there is no direct way to create an instance of it. As a workaround, `Java.newObject()` is provided.

As you also notice, `x10.lang.Comparable[T]` is mapped to `java.lang.Comparable` (cf. raw type). This is needed to map `x10.lang.String`, which implements `Comparable[String]`, to `java.lang.String` for performance, but as a trade off, this mapping results in the loss of runtime type information for `Comparable[T]` in Managed X10. The runtime of Managed X10 has built-in knowledge for `j.l.String`, but for other Java classes that implement `java.lang.Comparable`, `instanceof Comparable[Int]` etc. may return incorrect results. In principle, it is impossible to map X10 generic type to the existing Java generic type without losing runtime type information.

Fields As shown in Figure 8, instance fields of X10 classes and structs are translated to the instance fields of the generated Java classes. Static fields of X10 classes and structs are translated to the static methods of the generated Java classes, whose name has `get$` prefix. X10 does not have mutable static fields. Static fields of X10 interfaces are translated to the static methods of the special nested class named `$Shadow` of the generated Java interfaces.

Methods As shown in Figure 9, methods of X10 classes or structs are translated to the methods of the generated Java classes. Methods of X10 interfaces are translated to the methods of the generated Java interfaces.

[6] In a future version of X10, we plan to mostly eliminate the need for inserting boxing and unboxing manually by using standard Java wrapper classes such as `java.lang.Integer` and auto-boxing by `javac`.

X10		Java primitive (primary)		wrapper class
x.l.Byte	1y	byte	(byte)1	x.c.Byte
x.l.UByte	1uy	byte	(byte)1	x.c.UByte
x.l.Short	1s	short	(short)1	x.c.Short
x.l.UShort	1us	short	(short)1	x.c.UShort
x.l.Int	1	int	1	x.c.Int
x.l.UInt	1u	int	1	x.c.UInt
x.l.Long	1l	long	1l	x.c.Long
x.l.ULong	1ul	long	1l	x.c.ULong
x.l.Float	1.0f	float	1.0f	x.c.Float
x.l.Double	1.0	double	1.0	x.c.Double
x.l.Char	'c'	char	'c'	x.c.Char
x.l.Boolean	true	boolean	true	x.c.Boolean

Table 1. X10 types that have two representations in Java (`x10`, `lang`, `core` are represented as `x`, `l`, `c` respectively)

X10	Java
x.l.Any	j.l.Object
x.l.Comparable[T]	j.l.Comparable (raw type)
x.l.String	j.l.String
x.l.CheckedThrowable	j.l.Throwable
x.l.CheckedException	j.l.Exception
x.l.Exception	j.l.RuntimeException
x.l.ArithmeticException	j.l.ArithmeticException
x.l.ClassCastException	j.l.ClassCastException
x.l.IllegalArgumentException	j.l.IllegalArgumentException
x.l.IllegalStateException	j.l.IllegalStateException
x.u.NoSuchElementException	j.u.NoSuchElementException
x.l.NullPointerException	j.l.NullPointerException
x.l.NumberFormatException	j.l.NumberFormatException
x.l.UnsupportedOperationException	j.l.UnsupportedOperationException
x.l.IndexOutOfBoundsException	j.l.IndexOutOfBoundsException
x.l.ArrayIndexOutOfBoundsException	j.l.ArrayIndexOutOfBoundsException
x.l.StringIndexOutOfBoundsException	j.l.StringIndexOutOfBoundsException
x.l.Error	j.l.Error
x.l.AssertionError	j.l.AssertionError
x.l.OutOfMemoryError	j.l.OutOfMemoryError
x.l.StackOverflowError	j.l.StackOverflowError
x.l.InternalError	j.l.InternalError
void	void

Table 2. X10 types that are mapped to Java types. (`x10`, `java`, `lang`, `util` are represented as `x`, `j`, `l`, `u` respectively)

```
class C {
  static val a:Int = ...;
  var b:Int;
}
interface I {
  val x:Int = ...;
}
```

```
class C {
  static int get$a() { return ...; }
  int b;
}
interface I {
  abstract static class $Shadow {
    static int get$x() { return ...; }
  }
}
```

Figure 8. X10 fields in Java

```
interface I {
  def f():Int;
  def g():Any;
}
class C implements I {
  static def s():Int = 0;
  static def t():Any = null;
  def f():Int = 1;
  def g():Any = null;
}
```

```
interface I {
  int f$0();
  java.lang.Object g();
}
class C implements I {
  static int s$0() { return 0; }
  static java.lang.Object t() { return null; }
  int f$0() { return 1; }
  java.lang.Object g() { return null; }
}
```

Figure 9. X10 methods in Java

Every method whose return type has two representations, such as the types in Table 1, will have `$0` suffix with its name. For example, `def f():Int` in X10 will be compiled as `int f$0()` in Java. This is kind of optimization for the implementation of covariant method override while preserving the ability for passing and returning Java primitives without boxing. For details, please refer to our paper [16].

Constructors Constructors of X10 classes or structs are translated to the constructors of the generated Java classes.

Arrays X10 array type `x10.array.Array[T]` is a generic X10 type which is defined over a region that is made of a set of points. It is not easy to access an arbitrary X10 array in Java, but if we assume a dense, zero based, one dimensional X10 array, it is possible in the following way.

`x.a.Array[T]` is translated to a Java generic class `x10.array.Array<T>` which has an instance method `raw()` that returns an instance of a generic X10 type `x10.util.IndexedMemoryChunk[T]` which represents a dense, zero based, one dimensional X10 array. With `@NativeRep` annotation, `x.u.IndexedMemoryChunk[T]` is represented with a generic Java class `x10.core.IndexedMemoryChunk<T>` which has instance methods such as `getIntArray()` to get the Java backing array of type `int[]`.

X10's distributed array type `x10.array.DistArray[T]` has the same instance method `raw()` that returns an instance of `x.u.IndexedMemoryChunk[T]` which holds the Java backing array for the place where the program is running.

Exceptions X10 exception types except for the types in Table 2 are translated to Java exception types. Since all X10 exception types are subtype of `java.lang.RuntimeException`, Java programmers do not need to worry about how to handle them.

6. Related Work

Scala Scala [12] is a statically typed general purpose programming language for Java and .NET platforms. Like X10, Scala provides a unified type system, where all types are subtype of the single top type `scala.Any`. Scala supports value types. `scala.AnyVal` is the root class of all value types. Numbers, character, and boolean types are value types and they are mapped to Java primitive types.

Scala	Java
s.Any	j.l.Object
s.AnyRef	j.l.Object
s.Throwable	j.l.Throwable
s.Byte	byte
s.Short	short
s.Int	int
s.Long	long
s.Float	float
s.Double	double
s.Char	char
s.Boolean	boolean

Table 3. Scala types that are mapped to Java types. (scala, java, lang are represented as s, j, l respectively)

Since Scala 2.10 which was released in January 2013, Scala also supports user-defined value types.

When value types are upcast to either AnyVal, Any or T, they are boxed. To eliminate the cost of boxing, Scala uses an annotation at declaration site, in a form of [@specialized(scala.Int) T], to specify Scala compiler to generate specialized code for Int as well as general code for T.

Array type in Scala scala.Array[T] is a generic type of rank 1 which is specially handled by the compiler. For example, Array[Int] is represented as int[] etc. Methods of the Array[T] are implemented by implicit conversions to scala.collection.mutable.ArrayOps[T] and scala.collection.mutable.WrappedArray[T]. The conversion to ArrayOps[T] is temporary as all operations defined on ArrayOps[T] return an Array[T], while the conversion to WrappedArray[T] is permanent as all operations return a WrappedArray[T].

Scala does not have checked exceptions per se. Scala has a single exception hierarchy and its top type scala.Throwable is mapped to java.lang.Throwable for Java interoperability. Scala uses @throws[T <: Throwable] annotation to generate throws T clause.

Scala does not have its own string type and String is an alias of java.lang.String. Note that for Scala types in Table 3 are mapped to Java types.

Scala has overlapping goals with X10, providing a similarly rich type system with support for generic types and type composition. Scala's implementation uses type erasure in much the same way as Java, so it does not provide runtime access to reified generic types. However, Scala has two functions that may be used to provide substitute functionality in the user program directly. Manifest objects coupled with implicit method parameters can be used to pass the type information down to the generic method implementation. Scala compiler implicitly creates the manifest objects at the call site, where the concrete type information is available at the compilation time. Generic method implementation however, must declare implicit parameters to have access to them, and this implicit manifest parameter declaration must be repeated in all methods that may be called from the generic method called first. On the byte code level, manifest is explicitly created at caller site and is passed as ordinary parameter. In contrast, Managed X10 stores the type descriptor objects as instance fields of the generic object instances, so does not require type parameter arguments, except for constructor invocation and interface method calls. To reduce footprint, type descriptor objects are cached and reused for the same type parameter value.

Ceylon Ceylon [1] is a statically typed programming language for large-scale team development, addressing the problems of Java language that were discovered during years of using Java. Ceylon provides reified generics with reified types, declaration-site variance annotations and type bounds.

Ceylon	Java
c.l.Anything	j.l.Object
c.l.Object	j.l.Object
c.l.IdentifiableObject	j.l.Object
c.l.Null	j.l.Object
c.l.Boolean	boolean
c.l.Integer	long
c.l.Float	double
c.l.Character	int
c.l.String	j.l.String
c.l.Exception	c.l.Exception (in instantiations and extends clauses) j.l.Exception (in catch clauses) j.l.Throwable (everywhere else)
c.l.Sequence	c.l.Iterable

Table 4. Ceylon types that are mapped to Java types. (ceylon, java, lang(uage) are represented as c, j, l respectively)

Ceylon supports reified generics since M5 which was released in March 2013. Method overloading based on type parameter value is implemented by passing type descriptor as extra method arguments (i.e. type lifting). This is the same technique as X10 uses, but X10 even optimizes it to allow primitives in method parameters and return value [16].

Table 4 shows the list of special types that are mapped to Java types. As shown in the table, Ceylon has only one integer and one floating point types. For Java primitive types that are not directly mapped to Ceylon types, widening conversion will be applied.

Ceylon has a single exception hierarchy topped by ceylon.language.Exception. Ceylon does not have checked exceptions. Because in catch clause c.l.Exception is mapped to java.lang.Exception, there is no way to catch exceptions that are not subtype of java.lang.Exception, which include java.lang.Error and its subtypes.

Ceylon has Java array type ceylon.language.Array<T> which represents Java array type T[]. It is not possible at the moment to create an array of Java primitive types in Ceylon.

Kotlin Kotlin [9] is a statically typed programming language aimed at providing a pragmatic language for application development for JVM platform. Kotlin supports erasure based generics like Java. It also supports declaration-site variance.

Kotlin provides unified type system where numbers, character, and boolean types can instantiate type parameters. In Kotlin, numbers, character, and boolean types are mapped to Java primitive types and they are boxed as needed.

In Kotlin, array is a generic type kotlin.jet.Array<T>. However, unlike Java or X10, array is invariant. Unlike in X10, the backing array of generic array type Array<T> is always the array of reference type T[]. To avoid boxing/unboxing operations, Kotlin also has special non-generic array types for numbers, character, and boolean types such as kotlin.jet.IntArray etc. Table 5 shows the list of special types that are mapped to Java types.

Kotlin does not have checked exceptions. Compiler generates code that catches checked exception and converts it to unchecked exception.

Fantom Fantom [3] is a statically typed, general purpose, object-oriented programming language that runs on the Java Runtime Environment (JRE), JavaScript, and the .NET Common Language Runtime (CLR). Its primary design goal is to provide a standard library API that abstracts away the question of whether the code will ultimately run on the JRE or CLR.

Kotlin	Java
k.j.Any	j.l.Object
k.j.Byte	byte
k.j.Short	short
k.j.Int	int
k.j.Long	long
k.j.Float	float
k.j.Double	double
k.j.Char	char
k.j.Boolean	boolean
k.j.ByteArray	byte[]
k.j.ShortArray	short[]
k.j.IntArray	int[]
k.j.LongArray	long[]
k.j.FloatArray	float[]
k.j.DoubleArray	double[]
k.j.CharArray	char[]
k.j.BooleanArray	boolean[]

Table 5. Kotlin types that are mapped to Java types. (`kotlin`, `jet`, `java`, `lang` are represented as `k`, `j`, `j`, `l` respectively)

Fantom	Java
sys::Obj	j.l.Object
sys::Bool	boolean
sys::Int	long
sys::Float	double
sys::Str	j.l.String
sys::Decimal	j.m.BigDecimal
Foo[] (sys::List parameterized with Foo)	Foo[]
[java]fanx.interop::ByteArray	byte[]
[java]fanx.interop::ShortArray	short[]
[java]fanx.interop::IntArray	int[]
[java]fanx.interop::LongArray	long[]
[java]fanx.interop::FloatArray	float[]
[java]fanx.interop::DoubleArray	double[]
[java]fanx.interop::CharArray	char[]
[java]fanx.interop::BooleanArray	boolean[]
[java]foo.bar::Baz	foo.bar.Baz

Table 6. Fantom types that are mapped to Java types. (`java`, `lang`, `math` are represented as `j`, `l`, `m` respectively)

Fantom provides unified type system where all types are subtype of `sys::Obj`. Fantom supports value types `sys::Bool`, `sys::Int`, and `sys::Float` and they are mapped to Java primitive types `boolean`, `long`, and `double`, respectively. No user-defined values type are supported. When value types are coerced to/from either reference types or `T`, the compiler will generate boxing/unboxing operations. Fantom does not support user-defined generic types, however it has a set of built-in generic types `sys::List`, `sys::Map`, and `sys::Func`.

Table 6 shows the list of special types that are mapped to Java types. As shown in the table, Fantom provides separate Java array type for each Java primitive type.

An exception is a normal object which subclasses from `sys::Err`. Fantom does not use Java styled checked exceptions. Checked exceptions are caught at their call site and wrapped with `sys:Err` or its subtypes such as `sys::IOErr` and `sys::InterruptedErr`.

Xtend Xtend [18] is a statically typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language but improves on many aspects, which include extension methods, lambda expressions, operator overloading, type based switching, multiple dispatch, template expressions, statement expressions, properties, local type inference etc. Xtend fully supports Java generics.

Unlike other JVM languages, Xtend uses Java's type system without modifications. This guarantees that Xtend programmers will not run into any interoperability caveats. Integration with Java works as expected in both directions and the generated code runs as fast as or faster than hand-written Java.

7. Conclusion

In this paper we explained how Java Interoperability in Managed X10 is designed and implemented.

For multi-platform language such as X10, design of cross-language interoperability is more difficult than other JVM languages. Use of target-language specific types such as Java primitive types and Java array types can be easily implemented by introducing corresponding types in the source language. However, exception handing has different requirements. For portable exception handling in the source language, we need to have a single top exception type in the source language, which covers exceptions from both the source and all the target languages. To this end, X10 redesigned its type system in X10 2.3 and combined X10, Java, and C++ exception hierarchies into one.

Acknowledgments

We would like to thank past and current members of X10 team, namely, Igor Peshansky, Olivier Tardieu, Avraham Shinnar, Benjamin Herta, Mandana Vaziri, Kiyokuni Kawachiya, Akihiko Tozawa, Tatsuhiro Chiba, Salikh Zakirov, Yuki Makino, Takao Moriyama, and Tamiya Onodera for their valuable comments on the early design of Java interoperability.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

References

[1] Ceylon - A JVM Based Programming Language. `http://ceylon-lang.org/`

[2] CUDA C Programming Guide, Version 3.2. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`

[3] Fantom Programming Language. `http://fantom.org/`

[4] Apache Hadoop. `http://hadoop.apache.org/`

[5] JDT Core Component. `http://www.eclipse.org/jdt/core/index.php`

[6] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Publishing Company. 2013.

[7] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. A performance model for X10 applications: what's going on under the hood? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[8] K. Kawachiya, M. Takeuchi, S. Zakirov, and T. Onodera. Distributed Garbage Collection for Managed X10, In *Proceedings of the ACM SIGPLAN 2012 X10 Workshop (X10 '12)*, 2012.

[9] Kotlin - A New Programming Language for the JVM. `http://kotlin.jetbrains.org/`

[10] PGAS – Partitioned Global Address Space Languages. `http://www.pgas.org/`

[11] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`

[12] The Scala Programming Language. `http://www.scala-lang.org/`

[13] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased Performance for In-Memory Hadoop Jobs, In *Proceedings of the VLDB Endowment (VLDB '12)*, pp. 1736–1747, 2012.

[14] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Publishing Company. 1997.

[15] M. Takeuchi, Y. Makino, K. Kawachiya, H. Horii, T. Suzumura, T. Suganuma, and T. Onodera. Compiling X10 to Java, In *Proceedings of the ACM SIGPLAN 2011 X10 Workshop (X10 '11)*, 2011.

[16] M. Takeuchi, S. Zakirov, K. Kawachiya, and T. Onodera. Fast Method Dispatch and Effective Use of Primitives for Reified Generics in Managed X10, In *Proceedings of the ACM SIGPLAN 2012 X10 Workshop (X10 '12)*, 2012.

[17] X10: Performance and Productivity at Scale. `http://x10-lang.org/`

[18] Xtend - Modernized Java. `http://www.eclipse.org/xtend/`

A. Compilation and Execution

Compiling and executing X10 programs To compile and run X10 program that interoperates with existing Java code, the same `x10c` and `x10` commands are used with additional `-classpath` (or in short, `-cp`) option to specify the location of Java class files or jar files that your X10 program refers to.

```
x10c -cp JavaLib.jar -o X10Prog.jar X10Prog.x10
x10 -cp X10Prog.jar:JavaLib.jar X10Prog
```

Compiling Java programs To compile Java program that interoperates with X10 code, it is needed to specify the location of class files or jar files that were generated by `x10c` command. In addition, in many cases it is also needed to specify the location of standard X10 library jar file `x10.jar`. `x10c` command uses JDT Core Batch Compiler (a.k.a. Eclipse Compiler for Java) [5] as post compiler and generates Java 6 compliant class files.

For convenience, `x10cj` command, which accepts the same options as `javac` accepts and invokes the post compiler with the same options as `x10c` command passes, is provided.

```
x10cj -cp X10Lib.jar JavaProg.java
```

Executing Java programs The generated Java code by `x10c` command assumes that the runtime of Managed X10 is set up properly at each place before execution.

For X10 program that has `static def main(Rail[String]):void` method, `x10c` command generates a static nest class `$Main` with `static void main(java.lang.String[])` method. `x10` command executes the Java main method of the static nested class, which will first set up X10 runtime then execute the X10 main method.

To launch Java class that has `static void main(java.lang.String[])` method, a special launcher command `runjava`, which accepts the same options as `x10` command does, is provided. It will set up X10 runtime then execute the Java main method. All Java programs that interoperate with X10 code should be launched with it, and no other mechanisms, namely the direct execution with `java` command, are supported.

```
runjava -cp X10Lib.jar JavaProg
```