Fast Method Dispatch and Effective Use of Java Primitives for Reified Generics in Managed X10

2012/06/14
Mikio Takeuchi, Salikh Zakirov*, Kiyokuni Kawachiya, Tamiya Onodera
IBM Research - Tokyo

* Salikh Zakirov is currently affiliated with Google Inc.
This Talk Focuses on “Managed X10”

**Managed X10** = X10 implementation on Java (i.e. managed runtime)
- X10 program is translated into Java source code and compiled into Java bytecode, then executed on multiple Java VMs
Challenges in compiling X10 to Java with performance

We focus on primitive and generic types here.

- X10 does not have distinct primitive types, such as int and Integer in Java.
  - Type hierarchy

  ![Type Hierarchy Diagram]

  - Generics in X10 is based on reification, but in Java, it is based on erasure.
    - e.g. In X10, class C can implements both I[Int] and I[Double]

  ➔ How can we generate Java codes without degrading performance?
Numbers, character and boolean in Java

- In Java, numbers, character and boolean are primitives types (e.g. int).
  - Their values and operations are directly mapped to processor’s registers and instructions, therefore they are as fast as its native performance.
- Java primitives are *not* subtype of java.lang.Object.
  - They cannot be used to instantiate generic type parameter T.
  - Corresponding wrapper (boxed) classes are prepared for that purpose.
  - Primitives and wrapper classes are distinct types, although auto-boxing is supported by javac.

```
T t;
int sum = 0;
for (int i = 0; i < 10; ++i) sum += i;
t = sum;  // t = Integer.valueOf(sum);
```

Blue means primitive (int)
Red means wrapper class (Integer)
Integer.valueOf(int) is a Java library

Java type system
These are distinct types.
Numbers, character and boolean in Managed X10

- In X10, there is no distinction between primitive and wrapper types.
  - Numbers are subtype of x10.lang.Any, and they can be used to instantiate generic type parameter T.
  - However, using wrapper classes for simple calculation imposes overhead.

- Managed X10 uses Java primitives for numbers wherever possible, and converts them to wrapper classes as needed.
  - x10.lang.Int is compiled into int, but sometimes into x10.core.Int.
  - Similar mechanism is used also for unsigned integer types.

```
var t:T;
var sum:Int = 0;
for (var i:Int = 0; i < 10; ++i) sum += i;
t = sum;  // t = x10.core.Int.$box(sum);
```

Blue means primitive (int)
Red means wrapper class (Int)
Int.$box(int) is an X10 runtime library.
Covariant and generic return types in X10

- In X10, numbers are subtype of x10.lang.Any and can be used to instantiate generic type parameter T.
  - Therefore, a method returning numbers can override or implement methods returning Any or T.
  - This is another situation where numbers need to be reference types.

```java
abstract class C {
    abstract def f():Any;
}
class D extends C {
    def f():Int = 1;
}

abstract class C[T] {
    abstract def g():T;
}
class D extends C[Int] {
    def g():Int = 2;
}
```

Compiled versions of these methods need to return a reference type (x10.core.Int), rather than a primitive type (int).
Managed X10 generates *specialized method* to the type which instantiated generic type parameter T (class D in the examples).

**Covariant return type**

```scala
abstract class C {
  abstract def f():Any;
}
class D extends C {
  def f():Int = 1;
}
```

**Generic return type**

```scala
abstract class C[T] {
  abstract def g():T;
}
class D extends C[Int] {
  def g():Int = 2;
}
```
Reified generics in Managed X10

- Java supports erased generics.
  - Type parameters are erased at compile time.

- X10 supports reified generics.
  - The value of type parameter is accessible at runtime.
  - Methods can be overloaded if argument’s type parameter differs.

- C++ implements reified generics by specializing types (i.e. templates).
  - A large increase of classes is a drawback of this technique.
  - We want to utilize Java generics as much as possible.

- Managed X10 uses *type lifting* technique.
  - Type descriptor is stored in an instance field.
  - Method overloading is implemented by passing type descriptor as additional argument(s), and *self-dispatching* to an appropriate method.
Self dispatching overloaded methods

interface I[T] {
  def f(T):Any;
}
class C implements I[Int], I[Double] {
  def f(Int) = null;
  def f(Double) = null;
}
val c = new C();
val ii:I[Int] = c;
val id:I[Double] = c;
ii.f(0);  // f(Int) is invoked
id.f(0.0);  // f(Double) is invoked

This is impossible in Java.

Generated Java

interface I<T> {
  Object f(Object a1, Type t1);// dispatch method
}
class C implements I {
  Object f(Object a1, Type t1) { // dispatch method
    if (t1.equals(Types.INT))
      return f(Int.$unbox(a1));
    if (t1.equals(Types.DOUBLE))
      return f(Double.$unbox(a1));
    throw new x10.lang.Error();
  }  
  Object f(int) { return null; }  // actual methods
  Object f(double) { return null; }  
}
C c = new C();
I<Int> ii = (I) c;
I<Double> id = (I) c;
ii.f(Int.$box(0), Types.INT);
id.f(Double.$box(0.0), Types.DOUBLE);
Eliminating the self-dispatching cost

Self-dispatching is significantly slower than virtual method invocation.  
– Since it is implemented in user-level Java code, as a sequence of comparison of type descriptor followed by invocation of corresponding actual method.

To eliminate the self-dispatching cost,
Managed X10 uses method-name and parameter mangling.
Method name and parameter mangling

**Name mangling for normal methods**

```java
interface I<T> {}
class C<T,U> {
def f(Any):void {}
def f(T):void {}  // type parameter
def f(U):void {}
def g(I<T>):void {}  // parameterized type
def g(I[Int]):void {}
def h(Int):void {}
def h(UInt):void {}  // unsigned type
}
```

```java
interface I<T> {}
class C<T,U> {
void f(Object) {}
void f__0C$$T(T) {}  // type parameter
void f__0C$$U(U) {}
void g__0$1C$$T$2(I) {}  // parameterized type
void g__0$1x10$lang$Int$2(I) {}
void h(int) {}
void h__0$u(int) {}  // unsigned type
}
```

**Parameter mangling for constructors**

```java
interface I<T> {}
class C<T,U> {
C(Type T, Type U) {...}
C(Type T, Type U, Object $dummy) {...}
C(Type T, Type U, T, __0C$$T) {...}  // type parameter
C(Type T, Type U, U, __0C$$U) {...}
C<Type T, Type U, I<T>, __0$1C$$T$2>{...}  // parameterized type
C<Type T, Type U, I<Int>, __0$1x10$lang$Int$2} {...}
C(Type T, Type U, int) {...}
C(Type T, Type U, int, __0$u) {...}  // unsigned type
abstract static class __0C$$T {}  // synthetic classes
abstract static class __0C$$U {}
abstract static class __0$1C$$T$2 {}
abstract static class __0$1x10$lang$Int$2 {}
abstract static class __0$u {}
}
```

**Generated Java**

```java
interface I<T> {}
class C<T,U> {
    void f(Object) {}
    void f__0C$$T(T) {}  // type parameter
    void f__0C$$U(U) {}
    void g__0$1C$$T$2(I) {}  // parameterized type
    void g__0$1x10$lang$Int$2(I) {}
    void h(int) {}
    void h__0$u(int) {}  // unsigned type
}
```

**Generated Java**

```java
interface I<T> {}
class C<T,U> {
def this() {}
def this(Any) {}
def this(T) {}  // type parameter
def this(U) {}
def this(I<T>) {}  // parameterized type
def this(I[Int]) {}
def this(Int) {}
def this(UInt) {}  // unsigned type
}
```

**Caller can directly invoke a specific mangled method if argument type is known.**
Returning primitives from dispatch method

- Return type of dispatch method is Object.
  - Because a single dispatch method may correspond to multiple actual methods.
  - Therefore, primitives need to be converted to wrapper classes for return.

→ To remove the redundant conversions, Managed X10 generates a special dispatch method for each primitive return type in addition to a standard dispatch method.
Self dispatching (a case with single dispatch method corresponds to multiple actual methods)

```java
interface I[T] {}
interface J[T] { def f(I[T]):Any; }
abstract class S implements J[Int] {
    abstract def f(I[Int]):Any;
}
interface K[T] { def f(I[T]):Int; }
interface L[T] { def f(I[T]):UInt; }
interface M[T] { def f(I[T]):Int; }
interface N[T] { def f(I[T]):void; }
interface O[T] { def f(I[T]):Any; }

class C extends S implements K[Int],L[Float],M[Any],N[UInt],O[Long] {
    def f(I[Int]) = 1;
    def f(I[Float]) = 2u;
    def f(I[Any]) = 3;
    def f(I[UInt]) {} 
    def f(I[Long]):Any = null;
}
```
Special dispatch methods to return primitives

- In addition to the general dispatch method, ...
- Special version of dispatch methods are also generated to return primitive types
  - This method is invoked if returned value is used as Int in a caller

### Interface Definitions

- `interface I<T> {}`
- `interface J<T> { Object f(I a1, Type t1); } // dispatch method`
- `interface S extends Ref implements J {
  // dispatch method
  Object f(I a1, Type t1) { return f__0$1x10$lang$Int$2((I) a1); }
  abstract Object f__0$1x10$lang$Int$2$O((I) a1); // bridge method
}

- `interface K<T> { int f$I(I a1, Type t1); } // special dispatch methods`
- `interface L<T> { int f$i(I a1, Type t1); }
  interface M<T> { int f$I(I a1, Type t1); }`
- `interface N<T> { void f$V(I a1, Type t1); }
  interface O<T> { Object f(I a1, Type t1); } // dispatch method`

### Class C Implementation

```java
class C extends S implements K, L, M, N, O {
  Object f(I a1, Type t1) { // dispatch method
    if (t1.equals(ParameterizedType.make(I.$RTT, Types.INT)))  // K
      return Int.$box(f__0$1x10$lang$Int$2$O((I) a1));
    if (t1.equals(ParameterizedType.make(I.$RTT, Types.FLOAT)))// L
      return UInt.$box(f__0$1x10$lang$Float$2$O((I) a1));
    if (t1.equals(ParameterizedType.make(I.$RTT, Types.ANY)))  // M
      return Int.$box(f__0$1x10$lang$Any$2$O((I) a1));
    if (t1.equals(ParameterizedType.make(I.$RTT, Types.UINT))) // N
      f__0$1x10$lang$UInt$2((I) a1); return null;
    if (t1.equals(ParameterizedType.make(I.$RTT, Types.LONG))) // O
      return f__0$1x10$lang$Long$2((I) a1); throw new x10.lang.Error();
  }
}
```

### Generated Java Code

```java
// special dispatch methods
int f$I(I a1, Type t1) {
  if (t1.equals(ParameterizedType.make(I.$RTT, Types.INT)))  // K
    return f__0$1x10$lang$Int$2$O((I) a1);
  if (t1.equals(ParameterizedType.make(I.$RTT, Types.ANY))) // M
    return Int.$box(f__0$1x10$lang$Any$2$O((I) a1));
  throw new x10.lang.Error();
}
```

```java
int f$i(I a1, Type t1) { return f__0$1x10$lang$Float$2$O((I)a1); }//L
void f$V(I a1, Type t1) { f__0$1x10$lang$UInt$2((I) a1); } // N
```

```java
void f$V(I a1, Type t1) {
  f__0$1x10$lang$UInt$2((I) a1); // bridge method
  return Int.$box(f__0$1x10$lang$Int$2$O(a1));
}
```

```java
int f__0$1x10$lang$Int$2$O((I) a1) { return 1; } // actual methods
int f__0$1x10$lang$Float$2$O((I) a1) { return 2; }
int f__0$1x10$lang$Any$2$O((I) a1) { return 3; }
void f__0$1x10$lang$UInt$2((I) a1) {}
Object f__0$1x10$lang$Long$2((I) a1) { return null; }
```
Evaluation

Effectiveness of the optimizations to self-dispatching

- Eliminating self-dispatching

  Execution time of DispatchSpeedTest (Fig. 8)

<table>
<thead>
<tr>
<th></th>
<th>Self dispatching</th>
<th>Mangling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>2,521 µs</td>
<td>138 µs</td>
</tr>
</tbody>
</table>

→ 95% reduction (self-dispatching cost)

- Eliminating redundant boxing/unboxing

  Execution time of TreeMapTest (Fig. 9)

<table>
<thead>
<tr>
<th></th>
<th>With standard dispatcher</th>
<th>With special dispatcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>2,427 ms</td>
<td>2,065 ms</td>
</tr>
</tbody>
</table>

→ 15% reduction (boxing/unboxing cost)
Conclusions

We used Java **primitives** wherever possible, and converts them to wrapper classes as needed.

- To return primitives even in covariant or generic-type return, we generate special methods.

We implemented reified **generics** based on type lifting and self-dispatching.

- To remove self-dispatching cost, we applied name or parameter mangling.
  - We achieved 95% *reduction* in method dispatching time.
- To remove redundant conversion of primitives, we generated specialized method to return primitives.
  - We achieved 15% *reduction* in execution time.

- Proposed techniques can be used to implement reified generics in Java or in other languages running on Java.