`ThisJava`: An Extension of Java with Exact Types

Hyunik Na[1] and Sukyoung Ryu[2]

[1] S-Core., Ltd.
hina@kaist.ac.kr
[2] Department of Computer Science, KAIST
sryu.cs@kaist.ac.kr

Abstract. We propose `ThisJava`, an extension of Java-like programming languages with exact class types and `This` types, to support more useful methods with more precise types. To realize the proposed approach, we provide an open-source prototype implementation of the language and show its practicality and backward compatibility. We believe that our system elucidates the long pursuit of an object-oriented language with "`This` types."

1 Introduction

Objects in object-oriented programming often provide methods whose signatures include the object types themselves. The type of a receiver e of a method invocation "$e.m(e')$" is always the enclosing object type that defines or inherits the method m, which we call the method's *owner type*. In addition, many useful methods have their owner types in their parameter types or return types. In this paper, we call such methods with their owner types in their signatures *This-typed methods*.

Researchers have proposed various approaches to support `This`-typed methods. While the traditional `This` type denotes the "declared" type of a receiver, the `This` type in this paper denotes the "run-time" type of a receiver. Because run-time types of method receivers are mostly not available at compile time due to subtyping, the existing `This` types denote inexact compile-time types. To precisely capture the exact run-time type at compile time, our type system generates fresh type variables for the unknown run-time types at type checking time. This new notion of `This` types serves an essential role in the safe coexistence of recursive types and subtyping-by-inheritance.

To realize the proposed approach in a programming language, we extend Java with exact class types and `This` types. To support more methods with `This`-typed formal parameters, we enhance the type system with named wildcards and exact type inference, and we also introduce exact class type matching. To support more methods with `This`-typed results, we provide virtual constructors. We implemented our extension of Java, `ThisJava`, using JastAddJ [5] and made it open to the public:

`http://plrg.kaist.ac.kr/research/software`

We describe how the new features are compiled to Java bytecode, and how the new features interact with the existing Java features.

C.-c. Shan (Ed.): APLAS 2013, LNCS 8301, pp. 233–240, 2013.

2 Main Features of `ThisJava`

We extend Java with new typing features and language constructs to support more uses of exact types by allowing more `This`-typed methods. Due to space limitations, we briefly summarize the main features here and refer the interested reader to our earlier work [10,9].

We introduce *exact class types* of the form `#C` (read as "exact `C`") for a class `C` to represent run-time exact types of objects, and a type variable implicitly declared in the definition of `C`, `This`. In `ThisJava`, a `This` type appearing in the definition of a class (or interface) `C` denotes the *run-time type* of the special variable `this` (whereas the traditional `This` types denote its *declared type*).

To describe the relationships between exact types, `ThisJava` provides a type annotation of the form `</X/>`, called *named wildcards*. Using named wildcards, programmers can state that two formal parameter types or the result type and a formal parameter type are the same run-time types. Also, `ThisJava` performs a type-flow analysis called *exact type inference* to collect more equality relationships between exact types. Exact type inference lessens the programmers' burden of using exact type annotations such as exact class types and named wildcards. Exact type inference traces the flows of run-time types through the chains of def-use pairs. In particular, it infers exact types in a flow-sensitive manner to make more precise judgements on the run-time type matches of expressions.

To enhance the expressive power taking advantage of `This` types, `ThisJava` provides *virtual constructors* to represent "generic factory methods," methods that have the `This` type as their return type, and generates and returns an object of the same run-time type as its receiver, even when it is inherited by a subclass. In addition to generic factory methods, virtual constructors are useful to reduce code duplication by implementing common tasks for object generation in a superclass and allowing subclasses to inherit or override them, unlike ordinary constructors. An invocation of a virtual constructor is dynamically dispatched based on the run-time type of the receiver and generates an object of the same run-time type as the receiver.

While the `This` type in this paper allows more `This`-typed methods than traditional `This` types because it denotes run-time exact types rather than compile-time inexact types, purely static approaches to typing invocations of `This`-typed methods may require more relaxation. To allow more `This`-typed methods, `ThisJava` provides a language construct, `classesmatch`, to compare run-time types:

```
classesmatch (x,y) { /* then-block */ ... }
else { /* else-block */ ... }
```

At run time, it checks whether the run-time types of two variables are identical or not. If they are, the execution continues to evaluate `then-block`, otherwise the execution continues to evaluate `else-block`. Because the semantics of `classesmatch` guarantees that the run-time types of two variables are the same in `then-block`, it allows `This`-typed method invocations on the variables in `then-block`. Because programmers use (inexact) class types in most cases to enjoy subtype polymorphism, checking exact class types by run-time tests with `classesmatch` would provide full flexibility in using `This`-typed methods.

3 Compilation of ThisJava

The ThisJava compiler generates class files that can run on JVMs [8]. In this section, we describe how the compiler translates new language features into Java bytecode.

Exact Types Because JVM does not support exact types, the ThisJava compiler converts exact types to class types that JVM can understand. We call this process *inexactization*, which is very similar to *type erasure* [7, Section 4.6] in Java. The inexactization process converts most of the exact types to class types as follows:

1. A This type in a class C definition is converted to C.
2. For a class D and a named wildcard X, types #D and D</X/> are converted to D.

The only exception to the rule 1 is to support overriding a method or a virtual constructor that has a This-typed formal parameter as follows:

1′. When the declared parameter type of a method or a virtual constructor is This, the This type is converted to Object and appropriate type casting is applied to the formal parameter so that the generated class files can pass the *verification process* [8, Section 4.9].

Virtual Constructors. The ThisJava compiler converts a virtual constructor definition into an ordinary constructor definition and two stub method definitions. The ordinary constructor definition has the same signature and body as the original virtual constructor except that any exact types are converted to class types as described above. The first stub method, vcStub0, has the same list of formal parameters as the original virtual constructor, packs the parameters into an array of Objects, and invokes the second stub method. Unless the class defining the virtual constructor is abstract, the second stub method, vcStub1, unpacks the packed parameters and invokes the ordinary constructor discussed above. Any virtual constructor invocation in the original source code is converted to the invocation of a vcStub0 method. For example, the following:

```
class C {
  int fi; Point fp;
  This(int i, Point p) { fi = i; fp = p; }
  This copy() { return new This(fi,fp); }
}
```

is converted to the following:

```
class C {
  int fi; Point fp;
  C(int i, Point p) { fi = i; fp = p; }
  C vcStub0(int i, Point p) {
    Object[] pack = new Object[] {Integer.valueOf(i), p};
    return vcStub1(pack);
  }
  C vcStub1(Object[] pack) {
    int i = ((Integer)pack[0]).intValue();
    Point p = (Point)pack[1];
    return new C(i,p);
  }
  C copy() { return vcStub0(fi,fp); }
}
```

If the class defining a virtual constructor is abstract, the generated `vcStub1` method is also abstract. When a class inherits a virtual constructor, the code conversion adds to the class an ordinary constructor which has the same signature as the inherited constructor and just calls its super-constructor. If the class is not abstract, the code conversion also redefines (overrides) the `vcStub1` method so that it calls the added constructor. Under our compilation strategy for virtual constructors, the `vcStub1` methods always have the same signature `vcStub1(Object[])` to simulate overriding of virtual constructors by overriding the `vcStub1` methods. Packing and unpacking of parameters are necessary to keep the uniform signature of the `vcStub1` methods.

Compilation of virtual constructor definitions actually require more strategies than discussed above to properly handle *inner classes* [3, Chapter 5] in Java. When a class defining or inheriting a virtual constructor is an inner class or a subclass of an inner class, the code conversion has to take enclosing objects [3, Section 5.2] and variables in an enclosing scope [3, Section 5.3] into account so that they can be correctly passed to the generated constructor and stub methods at run time. We omit the details of such compilation, but our `ThisJava` compiler implementation is publicly available.

Type Testing and Casting with Exact Types. `ThisJava` supports type testing and casting with exact class types:

```
... (o instanceof #Point) ...     // (1)
... (#Point) o ...                // (2)
```

Because `#Point` is an exact class type, when `o` has a `ColorPoint` object, a subclass of `Point`, at run time (1) should produce `false` and (2) should produce an exception. Thus, the bytecode generated for (1) is similar to the bytecode generated for the following:

```
... (o == null ? false : o.getClass() == Point.class) ...
```

and the bytecode generated for (2) is similar to the one generated for the following:[1]

```
... (o == null || o.getClass() == Point.class ? (Point) o
                        : throw new ClassCastException()) ...
```

`ThisJava` does not support type testing and casting with `This` types and types with named wildcards because they can be expressed by the `classesmatch` construct. For example, the following example is illegal in `ThisJava`:

```
Object o; ...
if (o instanceof This) {
  This t = (This) o;  /* do something with t */ ...
}
```

but the following is legal and does what the above intends:

```
Object o; ...
classesmatch (o, this) { /* do it with o */ ... }
```

[1] Note that it is not legal Java code but pseudo code for explanation. The `throw` statement is syntactically illegal.

4 Interactions with Existing Features of Java

4.1 Overloading

Java supports *overloading* [3, Section 2.8], which allows two or more methods visible in a scope to have the same name if they have different signatures. For an invocation of overloaded methods, the types of the actual arguments to the invocation may be compatible with two or more signatures of the overloaded methods. In such cases, a Java compiler chooses the most specific method among the applicable ones, if any. Otherwise, it signals an ambiguous invocation error at compile time.

ThisJava extends the Java rule for selecting the most specific method because This types and types with named wildcards may put additional run-time type match constraints on some of the arguments to a method invocation. The ThisJava compiler selects the most specific method for a method invocation as follows:

1. For every occurrence of This type, if any, in the signatures of applicable methods, replace it as follows:
   - if the declared type of the receiver is This, replace the This type with the class enclosing the method invocation;
   - if the declared type of the receiver is C</X/> for a class C and a named wildcard X, replace the This type with C; and
   - otherwise, replace the This type with the declared type of the receiver.
2. For every type with a named wildcard C</X/>, if any, in the signatures of applicable methods, replace it with C;
3. Among the applicable methods with the modified signatures by the above steps 1 and 2, select the most specific one by using the Java rule for selecting the most specific method [7, Section 15.12.2.5] extended with exact class types.
4. If the most specific method is determined, we are done. Otherwise, compare the run-time type match constraints of applicable methods as follows:
   - for two run-time type match constraints A and B, A is more specific than B if A is not equal to B and A includes every constraint in B.
5. If the most specific method is determined, we are done. Otherwise, the method invocation is ambiguous.

If a method has formal parameters declared with exact types, the ThisJava compiler *mangles* the method's name when it generates a class file to avoid possible conflicts between overloaded method signatures.

4.2 Arrays

ThisJava allows *exact array types* such as #C[], This[][], and C</X/>[], but with some limitations. Subtyping with exact array types is different from subtyping with Java array types. While Java array types are covariant in the sense that D[] is a subtype of C[] if D is a subtype of C, exact array types in ThisJava are not covariant; while the This type in the definition of C, C</X/>, and #C are subtypes of C, This[], C</X/>[], and #C[] are not subtypes of C[]. In general, covariant array types are unsound and JVM checks every assignment to an array at run time [13, Section 15.5].

ThisJava supports array creation of exact class types but not the other exact types, and it does not allow the root class Object to define a virtual constructor. While ThisJava supports 'new #C[10]', for example, and inexactizes it as discussed in Section 3, 'new This[10]' and 'new C</X/>[10]' are not allowed because they should produce arrays of different types depending on the run-time type of an object. Similarly, because any array type is a subtype of Object, if Object defines a virtual constructor then array types should inherit the virtual constructor, which should create arrays of different types depending on the run-time type of an object.

4.3 Generics and Wildcards

Since J2SE 5.0, Java supports type-parameterized classes and methods, known as *generics*, and it allows *type wildcards* (denoted by the question mark "?") to serve as type arguments for parameterized types [3, Chapter 11]. The new features of ThisJava work well with generics and wildcards of Java, but there are some restrictions on mixed uses of ThisJava's exact types and Java's generics and wildcards.

First, ThisJava does not allow an exact type as the declared upper bound of a type variable or a type wildcard. For example, all the following upper bounds are illegal:

```
class C<X extends #Point> { ...              // illegal
  class I<Y extends This> {...}              // illegal
  void m(Point</E/> p) {
    class L<Z extends Point</E/>> {...} ... // illegal
  }
}
```

Even if ThisJava allowed the above example, because no exact type has a proper subtype, the type variables X, Y and Z may be instantiated only by their respective upper bounds. Therefore, the above is nothing more than the following:

```
class C { /* #Point instead of X */ ...
  class I { /* C.This instead of Y */ ...}
  void m(Point</E/> p) {
    class L { /* Point</E/> instead of Z */ ...} ...
  }
}
```

Similarly, List<? extends #Point> denotes the same type as List<#Point>.

Unlike type variables, type wildcards may have lower bounds in Java. ThisJava allows This types and exact class types as lower bounds of type wildcards, but it does not allow types with named wildcards as lower bounds of type wildcards. For example, the following use of Point</X/> is not allowed:

```
class C { void mth(List<? super Point</X/>> l) {...} // illegal}
```

because there may exist multiple run-time types for the named wildcard. For example, if the above definition was allowed in ThisJava, in the following invocation of mth:

```
class C { void mth2(C c, List<Point> l) { ... c.mth(l); ... }}
```

Point</X/> may be #Point or #ColorPoint. Instead, the following definition:

```
class C { void mth2(C</X/> c, List<? super Point</X/>> l) {...}}
```

is legal because for any invocation of mth2, the named wildcard X is singly determined to the run-time type of the first argument of the invocation.

4.4 Type Checking Existing Java Programs

To see the backward compatibility of our ThisJava implementation, we compiled 7637 Java source files that comprise most of the standard class library from the OpenJDK 1.6 source code [12] using the ThisJava compiler. We found only one case that requires a user annotation to be compiled by the ThisJava compiler, but considering the diversity of the test files that we used, we believe that all valid Java programs (possibly with some annotations described below) are valid ThisJava programs.

The following example shows the exceptional case that needs a user annotation:

```
List<C> l = Arrays.asList(new C(...));
```

where the asList static method defined in the utility class java.util.Arrays takes a variable number of arguments of type T for a type variable T and returns a list of type List<T>. Because the invocation of asList does not explicitly provide a type to instantiate the type argument T, a compiler should infer it from the type of the actual argument new C(...), which is C. Then, the type of the right-hand side of the assignment is List<C> which is the declared type of the left-hand side, l. However, in ThisJava, the type of new C(...) is #C, which makes the type of the right-hand side List<#C>, which is not a subtype of the declared type of l. Thus, the assignment expression is not well typed in ThisJava. However, ThisJava can accept it with an explicit type instantiation rather than depending on the type inference as follows:

```
List<C> l = Arrays.<C>asList(new C(...));
```

5 Related Work

One may use *F-bounded polymorphism* [4] to define binary methods. In a class (or interface) definition of C<X>, the type variable X acts like a This type because it denotes a concrete class that extends C<X>. However, unlike This types, X is not the type of the special variable this; the type of this is C<X> which is not a subtype of X. Therefore, when programmers need an object of type X, they may need to declare an abstract method to use instead of this, as Altherr and Cremet [2] described.

An *abstract type* in Scala [1], which is a type member of a class (or trait) whose complete definition is deferred to its subclasses, may act like a This type. However, Scala still has the code duplication problem because it does not have virtual constructors, and the classes should declare and define the abstract type explicitly while the ThisJava classes do not declare or define a This type.

In Jx [11], a value-dependent type x.class denotes the exact type of an *immutable* variable x. Jx's this.class and x.class appear to be the same as our This and X, respectively, when x is declared with C</X/> for some C in our mechanism. Thus, Jx's type system and ours have comparable expressiveness in handling exact types. But Jx's constraint that the variable x in x.class should be immutable may be too restrictive.

In *Rupiah* [6], Foster introduced ThisClass constructors. However, while a subclass in ThisJava may inherit its superclass' virtual constructor, a subclass in Rupiah should always override its superclass' ThisClass constructor. Similarly, Saito and Igarashi [14] propose *nonheritable methods*, which may not be inherited but should always be overridden by subclasses. In the definition of a nonheritable method whose

return type is `This`, the `This` type is considered as a supertype of the enclosing class so that the method can use an ordinary constructor to generate a return value. On the contrary, by allowing inheritance of a superclass' virtual constructor, `ThisJava` reduces code duplication of virtual constructors.

6 Conclusion

We have presented `ThisJava`, an extension of Java with exact types, virtual constructors, and run-time type matches, to support more useful methods with more precise types than the traditional object-oriented languages with `This` types. We describe the compilation strategies and the interaction between the new features and existing features such as overloading, covariant array types, and generics. We believe that `ThisJava` elucidates the long pursuit of an object-oriented language with `This` types by providing a flexible type system and language features and a practical open-source prototype implementation. Our future work includes rewriting of the standard class library using `This` types to experiment its practicality and usability.

Acknowledgments. This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2011-0016139 and NRF-2008-0062609).

References

1. Scala home, `http://www.scala-lang.org/`
2. Altherr, P., Cremet, V.: Adding type constructor parameterization to Java. In: 9th Workshop on Formal Techniques for Java-like Programs (2007)
3. Arnold, K., Gosling, J., Holmes, D.: The Java™ Programming Language, 4th edn. Addison-Wesley (August 2005)
4. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: FPCA (1989)
5. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, pp. 1–18 (2007)
6. Foster, J.: Rupiah: Towards an expressive static type system for java. Williams College Senior Honors Thesis (2001)
7. Gosling, J., Joy, B., Jr., Steele, G.L., Bracha, G.: The Java™ Language Specification, 3rd edn. Addison-Wesley (June 2005)
8. Lindholm, T., Yellin, F.: The Java™ Virtual Machine Specification, 2nd edn. Addison-Wesley (April 1999)
9. Na, H., Ryu, S.: A new formalization of subtyping to match subclasses to subtypes (June 2013), `http://plrg.kaist.ac.kr/_media/research/publications/resubtyping_report.pdf`
10. Na, H., Ryu, S., Choe, K.: Exact type parameterization and this Type support. In: TLDI (2012)
11. Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: OOPSLA (2004)
12. Oracle. Openjdk 6 source, `http://download.java.net/openjdk/jdk6/`
13. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
14. Saito, C., Igarashi, A.: Matching ThisType to subtyping. In: SAC (2009)