Genericity through Dependent Types

Nathaniel Nystrom *
nystrom@us.ibm.com

Vijay Saraswat *
vsaraswa@us.ibm.com

Abstract

Genericity is a key requirement for modern object-oriented languages. In this paper, we describe a design for generic types in the programming language X10. X10 has an expressive and powerful dependent type system in which types are specified by constraining the immutable state of objects. The immutable state of an object is represented by its *properties*, public final fields of the object. A *constrained type* then, is a defined by a class type and a boolean predicate on the properties of the class.

Generic types are defined in a natural extension of the dependent type system by first introducing *type properties* into the language, and then constraining those properties using subtyping constraints.

The type system presented here subsumes the expressive power of Java's generic types, virtual types, and generalized constraints proposed for C#. The system also admits an efficient implementation and eschews the pitfalls of a type erasure semantics. We describe also a local type inference algorithm for constrained types that permits type annotations and constraints to be elided by the programmer.

1. Introduction

X10 is a statically typed object-oriented language designed for high-performance computing [19]. The language extends a class-based sequential core language similar to Java with constructs for distribution and fine-grained concurrency. However, X10 does not yet support generic types, a standard feature of modern object-oriented languages. This paper presents a design for generics that is a natural extension of the language's core dependent type system.

The sequential semantics of X10 are similar to Java's X10 programs and execute on a Java virtual machine. After evaluating several existing proposals for generic types in Java-like languages [9, 23, 4, 17, 3, 21, 1, 2, 6, 7, 16], we concluded that these proposals were insufficient for our needs.

A problem with many of these proposals, and in particular with Java5 [9] and Scala [16], is that generic types are implemented via type erasure. Our design is not implemented via type erasure and, in addition, supports run-time introspection of generic types.

Another problem with many of these proposals is inadequate support for primitive types, especially arrays. The performance of primitive arrays is critical for the high-performance applications for which X10 is intended. These proposals introduce unnecessary boxing and unboxing of primitives. Our design does not require primitives be boxed.

The design of generics in X10 and is based on its existing dependent type system [19, 18]. To rule out large classes of errors statically, X10 provides *constrained types*, a form of dependent type defined on predicates over the immutable state of objects [19, 18]. The immutable state of an object is captured by its *value properties*: public final fields of the object. For instance, the following class declares a two-dimensional point with properties `x` and `y` of type `float`:

```
class Point(x: float, y: float) { }
```

A constrained type is a type `C{e}`, where `C` is a class and `e` is a boolean predicate on the properties of `C` and the final variables in scope at the type. For example, given the above class definition, the type `Point{x*x+y*y<1}` is the type of all points within the unit circle.

To support genericity these types are generalized to allow *type properties* and constraints on these properties. Like a value property, a type property is an instance member. The type properties of an object are bound to concrete types when the object is created. Types may be defined by constraining the type properties as well as the value properties of a class.

The following code declares a class `Cell` with a type property named `T`.

```
class Cell[T] {
    var x: T;
    def get(): T = x;
    def set(x: T) = { this.x = x: }
}
```

The class has a mutable field `x`, and has `get` and `set` methods for accessing the field.

This example shows that type properties are in many ways similar to type parameters as provided in object-oriented languages such as Java and Scala. Type properties are types in their own right: they may be used in any context a type may be used, including in `instanceof` and cast expressions. However, the key semantic distinction between type properties and type parameters is that type properties are instance members. Thus, for an expression `e` of type `Cell`, `e.T` is a type, equivalent to the concrete type to which `T` was initialized when the object `e` was instantiated. To ensure soundness, `e` is restricted to final access paths. Within the body of a class, a property name `T` resolves to `this.T` (or to `C.this.T` if `T` is a property of an enclosing class `C`), just as value properties are resolved.

As with value properties, type properties may be constrained by predicates to produce new types. X10 supports equality constraints, written T_1`==`T_2, and subtyping constraints, written T_1`<:`T_2. For instance, the type `Cell{T==String}` is the type of all `Cell`s that contain a `String`.

In general, the syntax of a constrained type is `C{c}`, where `C` is a base class and `c` is a predicate on the properties of `C`. For brevity, a constraint can be written as a comma-separated list of conjuncts; that is, the constraint `c1 && c2` can be written `c1, c2`.

Constraints on properties induce a natural subtyping relationship: `C{c}` is a subtype of `D{d}` if `C` is a subclass of `D` and `c` entails `d`.

We consider here only constraints on type properties. See Nystrom et al. [18] for a more thorough presentation of constrained types in X10. The following are legal types:

* IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights NY 10598 USA

- `Cell`. This type has no constraints on `T`. Any type that constrains `T`, those below, is a subtype of `Cell`. The type `Cell` is equivalent to `Cell{true}`. For a `Cell c`, the return type of the `get` method is `c.T`. Since the property `T` is unconstrained, the caller can only assign the return value of `get` to a variable of type `c.T` or of type `Object`. In the following code, `y` cannot be passed to the `set` method because it is not known if `Object` is a subtype of `c.T`.

```
val x: c.T = c.get();
val y: Object = c.get();
c.set(x); // legal
c.set(y); // illegal
```

- `Cell{T==float}`. The type property `T` is bound to `float`. Assuming `c` has this type, the following code is legal:

```
val x: float = c.get();
c.set(1.0);
```

  The type of `c.get()` is `c.T`, which is equivalent to `float`. Similarly, the `set` method takes a `float` as argument.

- `Cell{T<:int}`. This type constrains `T` to be a subtype of `int`. All instances of this type must bind `T` to a subtype of `int`. The following expressions have this type:

```
new Cell[int](1);
new Cell[int{self==3}](3);
```

  The cell in the first expression may contain any `int`. The cell in the second expression may contain only `3`. If `c` has the type `Cell{T<:int}`, then `c.get()` has type `c.T`, which is an unknown but fixed subtype of `int`. The `set` method of `c` can only be called with an object of type `c.T`.

- `Cell{T:>String}`. This type bounds the type property `T` from below. The `set` method may be called with any supertype of `String`; the return type of the `get` method is known to be a supertype of `String` (and implicitly a subtype of `Object`).

For brevity, the constraint may be omitted and interpreted as `true`. The syntax `C[T1,...,Tm](e1,...,en)` is sugar for `C{X1==T1,...,Xm==Tm,x1==e1,...,xn==en}` where `Xi` are the type properties and `xi` are the value properties of `C`. If either list of properties is empty, it may be omitted.

In this shortened syntax, a type argument `T` used may also be annotated with a *use-site variance tag*, either `+` or `-`: if `X` is a type property, then the syntax `C[+T]` is sugar `C{X<:T}` and `C[-T]` is sugar `C{X:>T}`; of course, `C[T]` is sugar `C{X==T}`.

The rest of the paper...

2. Overview of X10 syntax

Before describing the generic type system, we present an overview of X10's syntax and semantics. X10 is a class-based language similar to Java or Scala. Superficially, the language may be thought of as sequential Java with some elements of Scala syntax and with new constructs for concurrency and distribution. Like Java, the language provides both classes and interfaces; it does not yet support traits, as found in Scala.

Both classes and interfaces may define properties. Value properties may be considered to be public final fields. Whereas Java supports only static fields in interfaces, X10 allows interfaces to define value properties. Any class implementing an interface must declare, and initialize in its constructor, the properties inherited from the interface.

Classes may define fields, methods, and constructors. The declaration syntax, illustrated in the `Cell` example above, is similar to Scala's. Fields may be declared either `val` or `var`. A `val` is n *final* and must be assigned exactly once. Methods are declared with a `def` keyword. As in Java, methods may be declared `static`, however fields cannot. Constructor syntax is similar to method syntax and X10 adopts Scala's convention of using the name `this` for constructors. In X10, constructors have a return type, which constrains the properties of the new object.

class	L	::=	`class C[`$\overline{\texttt{X}}$`](`$\overline{\texttt{x}}$`: `$\overline{\texttt{T}}$`){c}`
			`  extends T {` $\overline{\texttt{K}}$ $\overline{\texttt{M}}$ $\overline{\texttt{F}}$ `}`
type	T	::=	`C`
		|	`e.X`
		|	`T{c}`
constructor	K	::=	`def this[`$\overline{\texttt{X}}$`](`$\overline{\texttt{x}}$`: `$\overline{\texttt{T}}$`){c}: T = e`
method	M	::=	`def m[`$\overline{\texttt{X}}$`](`$\overline{\texttt{x}}$`: `$\overline{\texttt{T}}$`){c}: T = e`
field	F	::=	`val x{c}: T = e`
		|	`var x{c}: T = e`
constraint	c	::=	`e`
expression	e	::=	`true`
		|	`x`
		|	$\texttt{e}_1$ `&&` $\texttt{e}_2$
		|	$\texttt{e}_1$ `==` $\texttt{e}_2$
		|	$\texttt{T}_1$ `<:` $\texttt{T}_2$
		|	$\texttt{T}_1$ `==` $\texttt{T}_2$
		|	...

Figure 1. X10 grammar

A subset of the grammar for X10 is shown in Figure 1. We elide features of the language not relevant to this paper. In the grammar $[\alpha]$ denotes an optional occurrence of the sequence of symbols α, α^* denotes zero or more occurrences of α, and α^+ denotes one or more occurrences of α.

3. Generic types

3.1 Use-site variance

3.2 Subtyping

`C{c}` is a subtype of `D{d}` if `C` is a subclass of `D` and `c` entails `d`.

4. Generic classes

Classes may be declared with any number of type properties and value properties. These properties can be constrained with a *class invariant*, specified by a `where` clause, a predicate on the properties of any instance of the class. The general form of a class definition is:

```
class C[X_1, ..., X_p](x_1: T_1, ..., x_k: T_k)
      where c
      extends B{c_0}
      implements I_1{c_1}, ..., I_n{c_n} {...}
```

4.1 Definition-site variance

In a class definition, a type property may be declared with a *definition-site variance tag*, either `+` or `-`. A `+` tag indicates that the class is covariant on the property; that is, given a definition `Cell[+T]`, if `A` <: `B`, then `Cell[A]` <: `Cell[B]`. Similarly, `Cell[-T]` indicates that `T` is contravariant in `Cell`; that is, if `A` <: `B`, then `Cell[B]` <: `Cell[A]`.

A definition-site variance tag changes the meaning of the syntactic sugar for the type `Cell[A]`. If the property is covariant (i.e., is declared as `+T`), `Cell[A]` is sugar for `Cell{T<:A}`. If the property is contravariant (`-T`), then `Cell[A]` is sugar for `Cell{T:>A}`.

Otherwise, the property is invariant and `Cell[A]` is sugar for `Cell{T==A}`.

The compiler should issue a warning if a covariant property is used in a negative position (e.g., in a method parameter type) in its class definition, or if a contravariant property is used in a positive position (e.g., in a method return type). Without these restrictions, methods or fields with types dependent on the property would be safe, but not be accessible using the default instantiation (e.g., `Cell[int]`).

4.2 Class invariant

4.3 Method parameters

Methods and constructors may have type parameters. For instance, the `List` class below defines a `map` method that maps each element of a list of `T` to a value of another type `S`, constructing a new list of `S`.

```
class List[T] {
    val array: Array[T];
    def map[S](f: T => S): List[S] {
        val newArray = new Array[S](array.length);
        for (i in [0:array.length-1]) {
            newArray(i) = f(array(i));
        }
        return new List(newArray);
    }
}
```

A parameterized method can is invoked by giving type arguments before the expression arguments. For example, the following code takes a list of `String`s and returns a list of string lengths of type `int`

```
xs: List[String] = ...;
ys: List[int] = xs.map[int]((x: String) => x.length());
```

4.4 Method where clauses

Method and constructor parameters, both value parameters and type parameters, can be constrained with a where clause on the method. For type parameters, this feature is similar to generalized constraints proposed for C# [7]. In the following code, the `T` parameter is covariant and so the `append` methods below are illegal:

```
class List[+T] {
   def append(other: T): List[T] = { ... }        // illegal
   def append(other: List[T]): List[T] = { ... }  // illegal
}
```

However, one can introduce a method parameter and then constrain the parameter from below by the class's parameter: For example, in the following code,

```
class List[+T] {
   def append[U](other: U): List[U] where T <: U = { ... }
   def append[U](other: List[U]): List[U] where T <: U = { ... }
}
```

The constraints must be satisfied by the callers of `append`. For example, in the following code:

```
xs: List[Number];
ys: List[Integer];
xs = ys; // ok
xs.append(1.0); // legal
ys.append(1.0); // illegal
```

the call to `xs.append` is allowed and the result type is `List[Number]`, but the call to `ys.append` is not allowed because the caller cannot show that `Number` <: `Double`.

4.5 Method overriding

Legal if any call to super method can call sub method.
covariant return contravaraint args weaker where clause

4.6 Constructor definitions

Constructors are defined using the syntax `def this`:

ConstructorDef ::= `def this` [[*TypeParameters*]] `(` [*Formals*] `)` [`:` [`where` *Constraint*]] `=` *Expression* `;`

Constructors must ensure that all properties of the new object are initialized and that the class invariants of the object's class and its superclasses and superinterfaces hold.

Properties are initialized with a `property` statement. For instance, the constructor for `Cell` ensures that the type property `T` is bound.

```
def this[T](x: T) = { property[T](); this.x = x; }
```

The `property` statement is used to set all the properties of the new object simultaneously; the syntax is similar to a `super` constructor call.

If the `property` statement is omitted, the compiler implicitly initailizes the properties from the formal type and value parameters of the constructor. The property statement for `Cell`'s constructor, for example, could have been omitted.

Constructors have "return types" that can specify an invariant satisfied by the object being constructed. The compiler verifies that the constructor return type and the class invariant are implied by the `property` statement and any `super` or `this` calls in the constructor body.

Classes that do not declare a constructor have a default constructor with a type parameter for each type property and a value parameter for each value property.

5. Discussion

5.1 Type properties versus type parameters

Type properties are similar, but not identical to type parameters. The differences may potentially confuse programmers used to Java generics or C++ templates. The key difference is that type properties are instance members and are thus accessible through access paths: `e.T` is a legal type.

Type properties, unlike type parameters, are inherited. For example, in the following code, `T` is defined in `List` and inherited into `Cons`. The property need not be declared by the `Cons` class.

```
class List[T] { }
class Cons extends List {
    def head(): T = { ... }
    def tail(): List[T] = { ... }
}
```

The analogous code for `Cons` using type parameters would be:

```
class Cons[T] extends List[T] {
    def head(): T = { ... }
    def tail(): List[T] = { ... }
}
```

We can make the type system behave as if type properties were type parameters very simply. We need only make the syntax `e.T` illegal and permit type properties to be accessible only from within the body of their class definition via the implicit `this` qualifier.

5.2 Wildcards

Wildcards in Java [9, 23] were motivated by the following example (rewritten in X10 syntax) from [23]. Sometimes a class needs a

field or method that is a list, but we don't care what the element type is. For methods, one can give the method a type parameter:

```
def aMethod[T](list: List[T]) = { ... }
```

This method can then be called on any `List` object. However, there is no way to do this for fields since they cannot be parameterized. Java introduced wildcards to allow such fields to be typed:

```
List<?> list;
```

In X10, a similar effect is achieved by not constraining the type property of `List`. One can write the following:

```
list: List;
```

Similarly, the method can be written without type parameters by not constraining `List`:

```
def aMethod(list: List) = { ... }
```

In X10, `List` is a supertype of `List[T]` for any `T`, just as in Java `List<?>` is a supertype of `List<T>` for any `T`. This follows directly from the definition of the type `List` as `List{true}`, and the type `List[T]` as `List{X==T}`, and the definition of subtyping.

Wildcards in Java can also be bounded. We achieve the same effect in X10 by using type constraints. For instance, the following Java declarations:

```
void aMethod(List<? extends Number> list) { ... }
<T extends Number> void aParameterizedMethod(List<T> list) { ... }
```

may be written as follows in X10:

```
def aMethod(list: List{T <: Number}) = { ... }
def aParameterizedMethod[T{self <: Number}](list: List[T]) = { ... }
```

Wildcard bounds may be covariant, as in the following example:

```
List<? extends Number> list = new ArrayList<Integer>();
Number num = list.get(0);      // legal
list.set(0, new Double(0.0)); // illegal
list.set(0, list.get(1));      // illegal
```

This can also be written in X10, but with an important difference:

```
list: List{T <: Number} = new ArrayList[Integer]();
num: Number = list.get(0);     // legal
list.set(0, new Double(0.0)); // illegal
list.set(0, list.get(1));      // legal! (when list is final)
```

Note because `list.get` has return type `list.T`, the last call in above is well-typed in X10; the analogous call in Java is not well-typed.

Finally, one can also specify lower bounds on types. These are useful for comparators:

```
class TreeSet[T] {
    def this[T](cmp: Comparator{T :> this.T}) { ... }
}
```

Here, the comparator for any supertype of `T` can be used as to compare `TreeSet` elements.

Another use of lower bounds is for list operations. The `map` method below takes a function that maps a supertype of the class parameter `T` to the method type parameter `S`:

```
class List[T] {
    def map[S](fun: Object{self :> T} => S) : List[S] = { ... }
}
```

5.3 Proper abstraction

Consider the following example adapted from [23]:

```
def shuffle[T](list: List[T]) = {
    for (i: int in [0..list.size()-1]) {
        val xi: T = list(i);
        val j: int = Math.random(list.size());
        list(i) = list(j);
        list(j) = xi;
    }
}
```

The method is parameterized on `T` because the method body needs the element type to declare the variable `xi`.

However, the method parameter can be omitted by using the type `list.T` for `xi`. Thus, the method can be declared with the signature:

```
def shuffle(list: List) { ... }
```

This is called *proper abstraction*.

This example illustrates a key difference between type properties and type parameters: A type property is a member of its class, whereas a type parameter is not. The names of type properties are visible outside the body of their class declaration.

In Java, Wildcard capture allows the parameterized method to be called with any `List`, regardless of its parameter type. However, the method parameter cannot be omitted: declaring a parameterless version of shuffle requires delegating to a private parameterized version that "opens up" the parameter.

5.4 Virtual types

Type properties share many similarities with virtual types [13, 12], particularly with sound formulations of virtual types using path-dependent types, as found in gbeta [8], Scala [16], and J& [15]. Constrained types are more expressive than virtual types since they can be constrained at the use-site, can be refined on a per-object basis without explicit subclassing, and can be refined contravariantly as well as covariantly.

Thorup [21] proposed adding genericity to Java using virtual types. For example, a generic `List` class can be written as follows:

```
abstract class List {
    abstract typedef T;
    void add(T element) { ... }
    T get(int i) { ... }
}
```

This class can be refined by bounding the virtual type `T` above:

```
abstract class NumberList extends List {
    abstract typedef T as Number;
}
```

And this abstract class can be further refined to *final bind* `T` to a particular type:

```
class IntList extends NumberList {
    final typedef T as Integer;
}
```

These classes are related by subtyping: `IntList` <: `NumberList` <: `List`. Only classes where `T` is final bound can be non-abstract.

In X10, an analogous `List` class would be written as follows:

```
class List[T] {
    def add(element: T) = { ... }
    def get(i: int): T = { ... }
}
```

```
expressions     e    ::=  ...
                     |   T has Sig
signatures      Sig  ::=  def this[X̄](x̄: T̄){c}: T
                     |   def m[X̄](x̄: T̄){c}: T
                     |   val x{c}: T
                     |   var x{c}: T
```

Figure 2. Grammar for structural constraints

`NumberList` and `IntList` can be written as follows:

```
class NumberList extends List{T<:Number} { }
class IntList extends NumberList{T==Integer} { }
```

However, note that X10's `List` is not abstract. Instances of `List` can instantiate `T` with a particular type and there is no need to declared classes for `NumberList` and `IntList`. Instead, one can use the types `List[+Number]` and `List[Integer]`.

Unlike virtual types, type properties can be refined contravariantly. For instance, one can write the type `List[-Integer]`, and even `List{Integer<:T, T<:Number}`.

6. Constraint system

7. Structural constraints

The type system is general enough to support not only subtyping constraints, but also structural constraints on types. The type system need not change except by extending the constraint system. The syntax for structural constraints is shown in Figure 2.

Structural constraints on types are found in many languages. Haskell [11] supports type classes. In Modula-3, type equivalence and subtyping are structural rather than nominal as in object-oriented languages of the C family such as C++, Java, Scala, and X10. The language PolyJ [3] allows type parameters to be bounded using structural where clauses, a form of F-bounded polymorphism [5]. For example, a sorted list class in PolyJ can be written as follows:

```
class SortedList[T] where T { int compare(T) } {
    void add(T x) { ... x.compare(y) ... }
}
```

The where clause states that the type parameter `T` must have a method `compare` with the given signatures.

To support this, X10 provides structural constraints on types. The analogous X10 code for `SortedList` is:

```
class SortedList[T] where T has compare(T): int {
    def add(x: T) = { ... x.compare(y) ... }
}
```

A structural constraint is of the form *Type* `has` *Signature*. A constraint is satisfied if the type has a member of the appropriate name and with a compatible type. The constraint `X has f(T1): T2` is satisfied by a type `T` if it has a method `f` whose type is a subtype of `(T1 => T2)`$[\mathtt{T}/\mathtt{X}]$. As an example, the constraint `X has equals(X): boolean` is satisfied by all three of the following classes:

```
class C { def equals(x: C): boolean; }
class D extends C { }
class E { def equals(x: Object): boolean; }
```

By using function types and where clauses on constructors, X10 can go further than PolyJ. Unlike in PolyJ, where the `compare` method must be provided by `T`, in X10 the `compare` function can be external to `T`. This is achieved as follows:

```
class SortedList[T] {
    val compare: (T,T) => int;
    def this(cmp: (T,T) => int) = { compare = cmp; }
    def add(x: T) = { ... compare(x,y) ... }
}
```

This permits `SortedList` to be instantiated using different compare functions:

```
val unixFiles    = new SortedList[String]
                       (String.compareTo.(String));
val windowsFiles = new SortedList[String]
                       (String.compareToIgnoreCase.(String
```

But, a problem with this approach is that the compare function must be provided to the constructor at each instantiation of `SortedList`. The problem can be resolved by using constructors with different structural constraints:

```
class SortedList[T] {
    val compare: (T,T) => int;
    def this[T]() where T has compareTo(T): int = {
        this[T](T.compareTo.(S));
    }
    def this[T](cmp: (T,T) => int) = { compare = cmp; }
    def add(x: T) = { ... compare(x,y) ... }
}
```

Now, `SortedList` can be instantiated with any type that has a `compareTo` method without explicltly specifying the method at each constructor call.

8. Type inference

Because constrained types can be verbose, X10 supports type inference to reduce the type annotation burden on the programmer.

The type inference algorithm allows types to be omitted altogether from many declarations and from method and constructor invocations. The algorithm also allows programmers to write a partially constrained type or just the base type in a declaration and to have a more precise constrained type inferred. For instance, it infers the type `int{self==3}` for the local variables `x`, `y`, `z` in the following code:

```
val x = 3;
val y: int = x;
val z: int{self>0} = x;
```

The algorithm is local: method and constructor parameter types, as well as the types of mutable fields, must be declared explicitly. For non-`private` members, this requirement is essential for enabling separate compilation. Limiting the scope of inference also eliminates one cause of potentially confusing error messages when types cannot be inferred.

The algorithm uses the subtyping constraint system described in Section **??**.

In general, an expression may have more than one satisfying type.

One requirement of the algorithm is that it report not only that there exists a satisfying assignment, but also reports the assignment itself. The algorithm chooses the most precise assignment.

Because methods can be overridden, the inferred return type may be too precise, preventing subclasses from overriding the method.

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : C\{c\} \\ \Gamma \vdash \mathit{mtype}(C,m) = [X_1,\ldots,X_k](x_1:T_1,\ldots,x_n:T_n)\{d\} \to T \\ \Gamma \vdash e_i : S_i \\ \Gamma \vdash_{\mathcal{C}} \exists \mathtt{this} : C\{c\}.\exists x_i : T_i.S_i <: T_i \wedge d\end{array}}{\Gamma \vdash e_0.m(e_1,\ldots,e_n) : T}$$

```
Types can be inferred if the constraints are satisfied.
Need to materialize the constraints.

1. union-find on equality constraints
2. solve the subtyping constraints -- collapse cycles into union-find
        if SCC has a contradiction, complain
3. materialize bounds

impl: use XConstraint for union-find

        - T <: S
        - whenever we ask if T <: S, ask TypeSystem, then add <:(T,S)
          to constraint if true
        - whenever we ask if T == S, ask TypeSystem, then add ==(T,S)
          to constraint if true

Based on Henglein, TAPOS 99

lower bound of X
        union type of types Ti with Ti --> X
upper bound of X
        interesection type of types Ti with X --> Ti

C{c} & D{d} = (C&D){c||d}
C{c} | D{d} = (C|D){c&&d}

C&D = gcd(C,D)
C|D = lca(C,D)

X <: int ||
X <: float
-->
X <: int|float
-->
X <: number

int <: X || float <: X
-->
int&float <: X
-->
void <: X

p==q || p==q  -->  p==q
p==q || p==r  -->  true
S<:T || S<:U  -->  S<:(T|U)
T<:S || U<:S  -->  (T&U)<:S
```

A key difference is that X10 supports where clauses that constrain method and constructor type and value arguments. The algorithm should infer not only the base type of a constrained type, but also the type and value constraints.

X10 should perform type inference of local variable types and of type arguments for method and constructor calls.

Consider the following method from [23]:

```
def choose[T](a: T, b: T): T { ... }
```

In the following snippet, the algorithm should infer the type `Collection` for `x`.

```
intSet: Set[int];
stringList: List[String];
val x = choose(intSet, stringList);
```

And in this snippet, the algorithm should infer the type `Collection[int]` for `y`.

```
intSet: Set[int];
intList: List[int];
val y = choose(intSet, intList);
```

Finally, in this snippet, the algorithm should infer the type `Collection{T <: Number}` for `z`.

```
intSet: Set[int];
numList: List{T <: Number};
val z = choose(intSet, numList);
```

The inference algorithm for Java 5 produces analogous results.

Now, consider the following example:

```
def union[T](a: Set[T], b: Set[T]) : Set[T];
```

The union method cannot be called with just arguments of type `Set`.

```
set1: Set;
set2: Set;
val a = union(set1, set2);
```

This is illegal because the type system cannot demonstrate that `set1.T` and `set2.T` are equal. The following, however, is acceptable:

```
set1: Set;
set2: Set[set1.T];
val a = union(set1, set2);
```

As another example from [23], consider the following method signature:

```
def unmodifiableSet[T](set: Set[T]): Set[T];
```

In Java, this method could be called with an argument of type `Set<?>`. This instantiates the method on `?`; that is, the wildcard is captured by the call, since any element type will be safe. A type variable can capture only one wildcard.

In X10, the method can be called with just a `Set` because there are no constraints on `T`. Using desugared syntax, the method is equivalent to:

```
def unmodifiableSet[T](set: Set{self.T==T}): Set{self.T==T}
```

Any `Set` can be passed in: for an argument `e`, the method is instantiated on `e.T`. Note that if this method were defined as:

```
def unmodifiableSet(set: Set): Set;
```

then the connection between the element types of the argument and of the return types would be broken. However, in X10, one could write use the following signature to keep the connection:

```
def unmodifiableSet(set: Set): Set[set.T];
```

9. Semantics

To illustrate the concepts behind constrained type-checking, we formalize the type system in a small calculus CFGJ based on Featherweight Java [10]. The calculus is an extension of Constrained Featherweight Java [18].

This section has not been checked carefully and probably contains many errors of omission and comission.

The language is functional in that assignment is not admitted. However, it is not difficult to introduce the notion of mutable fields, and assignment to such fields. Since constrained types may only refer to immutable state, the validity of these types is not compromised by the introduction of state. Further, we do not formalize overloading of methods. Rather, as with FJ, we simply require that the input program be such that the class name `C` and method name `m` uniquely select the associated method on the class. We do model properties, constrained clauses, class invariants, where clauses in methods and constructors, and dependent type casts.

Constraint term	t	::=	self | x | t.f | X | t.X | new C[$\overline{\texttt{T}}$]($\overline{\texttt{t}}$) | g[$\overline{\texttt{T}}$]($\overline{\texttt{t}}$)
Constraint	c, d	::=	true | p[$\overline{\texttt{T}}$]($\overline{\texttt{t}}$) | t = t | c, c | (x : T) c | [X] c
Class	L	::=	class C[$\overline{\texttt{X}}$]($\overline{\texttt{x}}$: $\overline{\texttt{T}}$) where c extends T {$\overline{\texttt{M}}$}
Method	M	::=	def m[$\overline{\texttt{X}}$]($\overline{\texttt{x}}$: $\overline{\texttt{T}}$) : T where c = e
Expression	e	::=	x | e.f | e.m[$\overline{\texttt{T}}$]($\overline{\texttt{e}}$) | new C[$\overline{\texttt{T}}$]($\overline{\texttt{e}}$) | e as T
Type	S, T, U	::=	C{d} | X | t.X

Figure 3. CFGJ Syntax

9.1 Constraint system

Constraints are assumed to be drawn from a fixed constraint system, C, with inference relation $\vdash_C$ [20]. All constraint systems are required to support the trivial constraint `true`, conjunction, existential quantification and equality on constraint terms. Constraint terms include `final` variables (including `this`), the special variable `self` (which may occur only in constraints `c` and not in expressions `e`), field selections `t.f`, type variables, and type selections `t.X`.

The syntax for the language is specified in Figure 3. In the syntax, "," binds tighter than ";". We use the syntax `(x : T) c` for the constraint obtained by existentially quantifying the variable `x` of type `T` in `c`, and the syntax `[X] c` for the constraint obtained by existentially quantifying the type variable `X` in `c`. `p` ranges over the collection of predicates supplied by the underlying constraint system, and `g` over the collection of functions.

A *class declaration* `class C`($\overline{\texttt{f}} : \overline{\texttt{T}}$) `where c extends D{d}` {$\overline{\texttt{M}}$} declares a class `C` with the fields $\overline{\texttt{f}}$ (of type $\overline{\texttt{T}}$), a *declared class invariant* `c`, a *superclass invariant* `d` and a collection of methods $\overline{\texttt{M}}$. The constraints `c` and `d` are true for all instances of the class `C` (this is verified in the rule for type-checking constructors, T-NEW). In these constraints, `this` may be used to reference the current object; `self` does not have any meaning and must not be used.

A *method declaration* `def m`[$\overline{\texttt{X}}$]($\overline{\texttt{x}} : \overline{\texttt{T}}$) : $\texttt{T}_0$ `where c` {...} specifies the type of the arguments and the result, as usual. The formal type parameters $\overline{\texttt{X}}$ and formal value parameters $\overline{\texttt{x}}$ may occur in the argument types $\overline{\texttt{T}}$ and the return type $\texttt{T}_0$. The constraint `c` specifies additional constraints on the parameters and on `this` that must hold for a method invocation to be legal. Note that `self` does not make sense in `c` (since no type is being defined), and must not occur in `c`.

A type is taken to be of the form `C{c}` where `C` is the name of a class or interface and `c` is a constraint; we say that `C` is the *base* of the type `C{c}`. We use the following shorthand for types: we write `(x : S) C{c}` for `C{(x : S) c}`, `[X] C{c}` for `C{[X] c}`, and `d,C{c}` for `C{d,c}`.

We denote the application of the substitution $\theta = [\overline{\texttt{t}}/\overline{\texttt{x}}]$ to a constraint `c` by $\texttt{c}[\overline{\texttt{t}}/\overline{\texttt{x}}]$. Application of substitutions is extended to types by: $\texttt{C\{c\}}\theta = \texttt{C\{c}\theta\texttt{\}}$.

We summarize here properties of constraint systems described in [20] that are needed for the proofs: constraint systems may be thought of as presented via an intuitionistic Gentzen proof system supporting identity; affine and exchange on the left; existential quantification and conjunction on the left and right; and closure under substitution of terms. All constraint systems are required to satisfy: $\texttt{new C}[\overline{\texttt{T}}](\overline{\texttt{t}}).\texttt{f}_i = \texttt{t}_i$ and $\texttt{new C}[\overline{\texttt{T}}](\overline{\texttt{t}}).\texttt{X}_i = \texttt{T}_i$ provided that $\mathit{fields}(\texttt{C}) = \overline{\texttt{f}} : \overline{\texttt{S}}$ (for some sequence of types $\overline{\texttt{S}}$) and $\mathit{types}(\texttt{C}) = \overline{\texttt{X}}$.

9.2 Static semantics

Typing judgments, shown in Figure 4 are of the form $\Gamma \vdash \texttt{e} : \texttt{T}$ where Γ is a multiset of type assertions `x : T`, kind assertions `X`, and constraints `c`. When Γ is empty, it is omitted.

A type assertion `x : C{c}` constrains the variable `x` to contain references to only those objects `o` that are instances of (subclasses of) `C` and for which the constraint `c` is true provided that occurrences of `self` in `c` are replaced by `o`. Thus in the constraint `c` of a constrained type `C{c}`, `self` may be used to reference the object whose type is being specified. Note that `self` is distinct from `this`: `this` is permitted to occur in the clause of a type `T` only if `T` occurs in an instance field declaration or instance method declaration of a class; as usual, `this` is considered bound to the instance of the class to which the field or method declaration applies.

The definition of *mtype*(C,m) (the signature of a method named `m` in class `C`), *mbody*(C,m), (the body associated with method `m` in type `C`) and *fields*(C) (the sequence of fields and their types inherited or defined at `C`) is essentially as specified in FJ [10] with the difference that the method of a signature is taken to be of the form $[\overline{\texttt{X}}](\overline{\texttt{x}} : \overline{\texttt{S}}) \rightarrow_{\texttt{c}} \texttt{T}$. The variables `x` and type variables `X` are permitted to occur in the types $\overline{\texttt{S}}$ and `T`, and in the constraint `c`, and are considered bound, and subject to alpha-renaming. The definitions of *mtype*, *mbody*, *fields* are extended to apply to constrained types by ignoring the constraint. For a substitution θ we define $\mathit{mtype}(\texttt{T},\texttt{m},\theta)$ as the signature obtained by applying θ to $\mathit{mtype}(\texttt{T},\texttt{m})$, renaming bound variables as necessary. Similarly, for a substitution θ we define $\mathit{fields}(\texttt{T},\theta)$ to be $\overline{\texttt{f}} : \overline{\texttt{S}}\theta$, if the sequence of inherited and defined fields of the class underlying the type `T` is $\overline{\texttt{f}} : \overline{\texttt{S}}$. We let $\mathit{fields}(\texttt{T},\texttt{x})$ stand for $\mathit{fields}(\texttt{T},[\texttt{x}/\texttt{self}])$.

We define $\sigma(\Gamma)$ to be the set of constraints obtained from Γ by replacing each type assertion $\texttt{x} : \texttt{C\{d\}}$ in Γ with $\texttt{d}[\texttt{x}/\texttt{self}], \mathit{inv}(\texttt{C},\texttt{x})$ and retaining any constraint in Γ.

T-VAR extends the identity rule ($\Gamma, x : C \vdash x : C$) of FJ to take into account the constraint entailment relation.

T-CAST encapsulates the three inference rules of FJ: T-UCAST, T-DCAST and T-SCAST for upwards cast, downwards cast, and "stupid" cast respectively.

This paragraph is out-of-date. In T-FIELD, we postulate the existence of a receiver object `o` of the given static type ($\texttt{T}_0$). $\mathit{fields}(\texttt{T}_0,\texttt{o})$ is the set of typed fields for $\texttt{T}_0$ with all occurrences of `this` replaced by `o`. We record in the resulting constraint that $\texttt{o.f}_i = \texttt{self}$.[1] This permits transfer of information that may have been recorded in $\texttt{T}_0$ about the field $\texttt{f}_i$.

This paragraph is out-of-date. Similarly, in T-INVK we postulate the existence of a receiver object `o` of the given static type. For any type T, object `o` of type T and method name `m`, let $\mathit{mtype}(\texttt{T},\texttt{m},\texttt{o})$ be a copy of the signature of the method with `this` replaced by `o`. We establish (under the assumption that the formals ($\overline{\texttt{z}}$) have the static type of the actuals)[2] that actual types are subtypes of the formal types, and the method constraint is satisfied. This permits us to record the constraint `d` on the return type, with the formal variables $\overline{\texttt{z}}$ existentially quantified.[3]

This paragraph is out-of-date. In T-NEW, similarly, we establish that the static types of the actual arguments to the constructor are subtypes of the declared types of the field, and contain enough information to satisfy the class invariant, `c`. The declared types (and `c`) contain references to $\texttt{this.}\overline{\texttt{f}}$; these must be replaced by the formals $\overline{\texttt{f}}$, which carry information about the static type of the actuals. Note that the object `o` we hypothesized in an analogous situation in T-INVK does not exist; it will exist on successful invocation of the constructor. The constrained clause of the `new` expression contains all the information that can be gleaned from the static types of the actuals by assigning them to the corresponding fields of the object being created.

[1] A new name `o` is necessary to name this object since `e` cannot be used. Arbitrary term expressions `e` are not permitted in constraints; the functions used in `e` may not be known to the constraint system, and `e` may have side-effects.

[2] This is stronger than assuming $\overline{\texttt{z}}$.

[3] Recall that the $\overline{\texttt{z}}$ may occur in `d` but must not occur in a type in the calling environment; hence they must be existentially quantified in the resulting constraint.

$$\frac{\sigma(\Gamma, \mathtt{x} : \mathtt{C}\{\mathtt{c}\}) \vdash_{\mathcal{C}} \mathtt{d}[\mathtt{x}/\mathtt{self}]}{\Gamma, \mathtt{x} : \mathtt{C}\{\mathtt{c}\} \vdash \mathtt{x} : \mathtt{C}\{\mathtt{d}\}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{U}}{\Gamma \vdash \mathtt{e\ as\ T} : \mathtt{T}} \quad \text{(T-CAST)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}_0 \quad \mathit{fields}(\mathtt{T}_0, \mathtt{z}_0) = \overline{\mathtt{f}} : \overline{\mathtt{U}} \quad (\mathtt{z}_0 \text{ fresh})}{\Gamma \vdash \mathtt{e.f}_i : ((\mathtt{z}_0 : \mathtt{T}_0)\ \mathtt{z}_0 = \mathtt{this}, \mathtt{z}_0.\mathtt{f}_i = \mathtt{self})\ \mathtt{U}_i} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \quad \Gamma \vdash \mathtt{T}_0 <: \mathtt{C} \\ \mathit{mtype}(\mathtt{C}, \mathtt{m}, \mathtt{x}_0) = [\overline{\mathtt{X}}](\overline{\mathtt{x}} : \overline{\mathtt{U}}) \to_{\mathtt{c}} \mathtt{U} \\ \Gamma' = \Gamma, \overline{\mathtt{X}}, \overline{\mathtt{X}} = \overline{\mathtt{S}}, \mathtt{x}_0 : \mathtt{T}_0, \mathtt{x}_0 = \mathtt{this}, \overline{\mathtt{x}} : \overline{\mathtt{T}} \\ \Gamma' \vdash \overline{\mathtt{T}} <: \overline{\mathtt{U}} \quad \sigma(\Gamma') \vdash_{\mathcal{C}} \mathtt{c} \quad (\mathtt{x}_0, \overline{\mathtt{x}}, \overline{\mathtt{X}} \text{ fresh})\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}[\overline{\mathtt{S}}](\overline{\mathtt{e}}) : [\overline{\mathtt{X}}]\ (\mathtt{x}_0, \overline{\mathtt{x}} : \mathtt{T}_0, \overline{\mathtt{T}})\ \mathtt{U}} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \\ \theta = [\mathtt{x}_0, \overline{\mathtt{x}}/\mathtt{this}, \mathtt{this}.\overline{\mathtt{f}}] \quad \mathit{fields}(\mathtt{C}, \theta) = \overline{\mathtt{f}} : \overline{\mathtt{U}} \\ \Gamma, \overline{\mathtt{X}}, \overline{\mathtt{X}} = \overline{\mathtt{S}}, \mathtt{x}_0 : \mathtt{C}, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{T}} <: \overline{\mathtt{U}} \\ \sigma(\Gamma, \mathtt{x}_0 : \mathtt{C}, \overline{\mathtt{x}} : \overline{\mathtt{T}}) \vdash_{\mathcal{C}} \mathit{inv}(\mathtt{C}, \theta)\end{array}}{\Gamma \vdash \mathtt{new\ C}[\overline{\mathtt{S}}](\overline{\mathtt{e}}) : \mathtt{C}\{(\overline{\mathtt{x}} : \overline{\mathtt{T}})\ \mathtt{self}.\overline{\mathtt{f}} = \overline{\mathtt{x}}\}} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Gamma = \overline{\mathtt{X}}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}, \mathtt{c} \\ \Gamma \vdash \mathtt{e} : \mathtt{U} \quad \Gamma \vdash \mathtt{U} <: \mathtt{T}\end{array}}{\mathtt{def\ m}[\overline{\mathtt{X}}](\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{U\ where\ c} = \mathtt{e}\ \text{OK in C}}$$

$$\frac{\overline{\mathtt{M}}\ \text{OK in C}}{\mathtt{class\ C}[\overline{\mathtt{X}}](\overline{\mathtt{f}} : \overline{\mathtt{T}})\ \mathtt{where\ c\ extends\ D}\{\mathtt{d}\}\ \{\overline{\mathtt{M}}\}\ \text{OK}}$$

$$\mathtt{C} <: \mathtt{C} \quad \text{(S-REFL)}$$

$$\frac{\mathtt{class\ C}[\overline{\mathtt{X}}](\overline{\mathtt{f}} : \overline{\mathtt{T}})\ \mathtt{where\ c\ extends\ D}\{\mathtt{d}\}\ \{\overline{\mathtt{M}}\}}{\mathtt{C} <: \mathtt{D}} \quad \text{(S-EXTENDS)}$$

$$\frac{\mathtt{C} <: \mathtt{D} \quad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}} \quad \text{(S-TRANS)}$$

$$\frac{\mathtt{C} <: \mathtt{D} \quad \sigma(\Gamma, \mathtt{x} : \mathtt{C}\{\mathtt{c}\}) \vdash_{\mathcal{C}} \mathtt{d}[\mathtt{x}/\mathtt{self}] \quad (\mathtt{x} \text{ fresh})}{\Gamma \vdash \mathtt{C}\{\mathtt{c}\} <: \mathtt{D}\{\mathtt{d}\}} \quad \text{(S-DEP)}$$

$$\frac{\Gamma \vdash_{\mathcal{C}} \mathtt{T} <: \mathtt{U}}{\Gamma \vdash \mathtt{T} <: \mathtt{U}} \quad \text{(S-CONSTRAINT)}$$

Figure 4. CFGJ semantics

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}} : \overline{\mathtt{U}}}{(\mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\overline{\mathtt{e}})).\mathtt{f}_i \to \mathtt{e}_i} \qquad \text{(R-FIELD)}$$

$$\frac{\vdash \mathtt{C} <: \mathtt{T}[\mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\overline{\mathtt{e}})/\mathtt{self}]}{\mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\overline{\mathtt{e}})\ \mathtt{as}\ \mathtt{T} \to \mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\overline{\mathtt{e}})} \qquad \text{(R-CAST)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{x}.\mathtt{e}_0}{(\mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}}) \to \mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\overline{\mathtt{e}})/\mathtt{this}]} \qquad \text{(R-INVK)}$$

$$\frac{\mathtt{e} \to \mathtt{e}'}{\mathtt{e}.\mathtt{f}_i \to \mathtt{e}'.\mathtt{f}_i} \qquad \text{(RC-FIELD)}$$

$$\frac{\mathtt{e} \to \mathtt{e}'}{\mathtt{e}\ \mathtt{as}\ \mathtt{T} \to \mathtt{e}'\ \mathtt{as}\ \mathtt{T}} \qquad \text{(RC-CAST)}$$

$$\frac{\mathtt{e}_0 \to \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \to \mathtt{e}_0{}'.\mathtt{m}(\overline{\mathtt{e}})} \qquad \text{(RC-INVK-RECV)}$$

$$\frac{\mathtt{e}_i \to \mathtt{e}_i{}'}{\mathtt{e}_0.\mathtt{m}(\ldots, \mathtt{e}_i, \ldots) \to \mathtt{e}_0.\mathtt{m}(\ldots, \mathtt{e}_i', \ldots)} \qquad \text{(RC-INVK-ARG)}$$

$$\frac{\mathtt{e}_i \to \mathtt{e}_i{}'}{\mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\ldots, \mathtt{e}_i, \ldots) \to \mathtt{new}\ \mathtt{C}[\overline{\mathtt{T}}](\ldots, \mathtt{e}_i', \ldots)} \qquad \text{(RC-NEW-ARG)}$$

Figure 5. CFGJ semantics

Let C be a class declared as

$$\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{T}})\ \mathtt{where}\ \mathtt{c}\ \mathtt{extends}\ \mathtt{D}\{\mathtt{d}\}\{\overline{\mathtt{M}}\}.$$

Let θ be a substitution and the type T be based on C. We define $\mathit{inv}(T, \theta)$ as the conjunction $\mathtt{c}\theta, \mathtt{d}\theta$ and (recursively) $\mathit{inv}(\mathtt{D}, \theta)$. We bottom out with $\mathit{inv}(\mathtt{Object}, \theta) = \mathtt{true}$. For a variable x, we use the shorthand $\mathit{inv}(\mathtt{C}, \mathtt{x})$ to mean $\mathit{inv}(\mathtt{C}, [\mathtt{x}/\mathtt{self}])$.

We add two subtyping rule to the rules of FJ. The first, S-DEP, requires that the subtype constraint entail the supertype constraint. Whenever we state an assumption of the form "x is fresh" in a rule we mean it is not free in the consequent of the rule. The second rule, S-CONSTRAINT, states that one type is a subtype of another if the constraint system says so.

The operational semantics are essentially those of FJ.

10. Implementation

This section describes a possible implementation approach that performs run-time instantiation of classes, similar to the implementation of parameterized classes in NextGen [1, 2]. We describe only the translation to Java; an obvious translation strategy for the C++ backend would use templates, but the details have not been worked out. Another possible translation, based on the implementation of PolyJ [3], uses *adapter objects* to allow generic code to invoke methods of instances of the its type properties.

The translation is described by example. A more thorough and complete description will be given after some experience with the actual implementation.

Consider the code in Figure 6. It contains most of the features of generics that have to be translated.

10.1 Eliminating method type parameters

The first step in translation is to remove method parameters by introducing generic inner class for each generic method. Constructor type parameters are not changed. In addition, function type `T => S` is translated to `Fun1[T,S]`. This step produces the code in Figure 7: At this point, the code consists only of generic classes. The remaining translation introduces a run-time representation for the type properties of these classes.

10.2 Run-time instantiation

In this translation the type properties are represented as instances of a `Type` class, analogous to `java.lang.Class`. Each generic class has a `Type`-typed field for each of its type properties initialized by the class's constructor. The `Type` objects are used to implement `instanceof` and cast operations.

```
interface Type {
    boolean instanceof$(Object x);
    <T> T cast$(Object x);
}
```

In this translation, which is partially based on the NextGen [1, 2] translation, a generic class is translated into a *base interface* and a *template class* that implements the base interface. At runtime, the first time a generic class is instantiated a class loader loads *template class*, rewriting the bytecode to instantiate the type properties as appropriate.

For example, the code for class C above is translated into the template class in Figure 8 with supporting classes Figure 9. When instantiating the template, the string "{0}" is substituted with the

```
class C[T] {
    var x: T;
    def this[T](x: T) { this.x = x; }
    def set(x: T) { this.x = x; }
    def get(): T { return this.x; }
    def map[S](f: T => S): S { return f(this.x); }
    def d() { return new D[T](); }
    def t() { return new T(); }
    def isa(y: Object): boolean { return y instanceof T; }
}

val x : C = new C[String]();
val y : C[int] = new C[int]();
val z : C{T <: Array} = new C[Array[int]]();
x.map[int](f);
new C[int{self==3}]() instanceof C[int{self<4}];
```

Figure 6. Code to translate

```
class C[T] where T has T() {
    var x: T;

    def this[T](x: T) { this.x = x; }

    def set(x: T) { this.x = x; }
    def get(): T { return this.x; }

    def d() { return new D[T](); }
    def t() { return new T(); }

    def isa(y: Object): boolean { return y instanceof T; }

    // Translation of map to an inner class
    class map$[T,S] {
        def apply(c: C[T], f: Fun1[T,S]) { return f(c.x); }
    }
}

val x : C = new C[String]();
val y : C[int] = new C[int]();
val z : C{T <: Array} = new C[Array[int]]();
new map$[x.T,int]().apply(x,f);
new C[int{self==3}]() instanceof C[int{self<4}];
```

Figure 7. After removing method parameters

name of the actual type property.[4] Since methods of `C` can be called in a context where the property instantiation is not known, each method in the template class has to be implemented twice: once with an Object interface and once with an instantiated interface.

We translate `instanceof` and cast operations to calls to methods of a `Type` because the actual implementation of the operation may require run-time constraint solving or other complex code that cannot be easily substituted in when rewriting the bytecode during instantiation.

11. Related work

[22] [23] [7] [14] [3] [2] [1] [12] [13] [21]

12. Conclusions

We have presented a preliminary design for supporting genericity in X10 using type properties. This type system generalizes the existing X10 type system. The use of constraints on type properties allows the design to capture many features of generics in languages like Java 5 and C# and then to extend these features with new more expressive power. We expect that the design admits an efficient implementation and intend to implement the design shortly.

[4] In a real implementation, the names would be mangled as appropriate.

```
class C{0} implements C {
    final Type T = {0}$Type.it;
    {0} x;
    C{0}({0} x) { this.x = x; }

    void set$(Object x) { set(({0}) x); }
    void set({0} x) { this.x = x; }

    Object get$() { return ({0}) get(); }
    {0} get() { return this.x; }

    D d$() { return d(); }
    D{0} d() { return new D{0}(); }

    Object t$() { return t(); }
    {0} t() { return new {0}(); }

    boolean isa(Object y) { return T.instanceof$(y); }

    static class map$Type extends Type {
        ...
        static map$Type instantiate$(Type T, Type S) { ... }
    }

    static class map$Type{0}{1} extends Map$Type {
        map$ new$() { return new map${0}{1}(); }
    }

    interface map$ {
        Object apply$(C c, Fun1 f);
    }

    class map${0}{1} implements map$ {
        final Type T = {0}$Type.it;
        final Type S = {1}$Type.it;
        Object apply$(C c, Fun1 f)
          { return apply((C{0}) c, (Fun1{0}{1}) f); }
        {1} apply(C{0} c, Fun1{0}{1} f) { return f(c.x); }
    }
}

C x = new C$String();
C$int y = new C$int();
C z = new C$Array$int();
C.map$Type.instantiate$(x.T, int$Type.it).new$().apply$(x,f);
C$int$self$lt$4.instanceof$(new C$int$self$eqeq$3());
```

Figure 8. Translation to Java

References

[1] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *OOPSLA*, pages 96–114, October 2003.

[2] Eric E. Allen and Robert Cartwright. Safe instantiation in Generic Java. Technical report, March 2004.

[3] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, 1997.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programing Language. In *OOPSLA*, 1998.

[5] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[6] ECMA. ECMA-334: C# language specification, June 2006. http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf.

[7] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for c# generics. In *ECOOP*, 2006.

[8] Erik Ernst. *gbeta: A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[9] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2006.

```
class C$Type implements Type {
    static Type it = new C$Type();
    boolean instanceof$(Object x) { return x instanceof C; }

    static Map<Type,Type> instantiations;

    static Type instantiate$(Type T) {
        instantiations.get(T);
    }
}

class C{0}$Type implements Type {
    static Type it = new C{0}$Type();
    boolean instanceof$(Object x) { return x instanceof C{0}; }
}

interface C {
    void set$(Object x);
    Object get$();
    D d$();
    Object t$();
    boolean isa$(Object y);
}
```

Figure 9. Translation to Java

[10] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1999.

[11] Haskell 98: A non-strict, purely functional language, February 1999. Available at `http://www.haskell.org/onlinereport/`.

[12] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[13] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented progr amming. pages 397–406, October 1989.

[14] Andrew Myers and Barbara Liskov. Efficient implementation of parameterized types despite subtyping. Technical Report Thor Note 9, MIT LCS, June 1994.

[15] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software extension. In *Proceedings of the 2006 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 21–36, Portland, OR, October 2006.

[16] Martin Odersky. Report on the programming language Scala. Technical report, EPFL, 2006.

[17] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. pages 146–159, Paris, France, January 1997.

[18] Nathaniel Nystrom Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 2008 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2008.

[19] V. Saraswat et al. Report on the programming language X10. Technical report, IBM T.J. Watson Research Center, 2006.

[20] Vijay Saraswat. The category of constraint systems is Cartesian closed. In *LICS '92*, pages 341–345, 1992.

[21] Kresten Krab Thorup. Genericity in Java with virtual types. Number 1241, pages 444–471, 1997.

[22] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining virtual types and parameterized classes. In *ECOOP*, 1998.

[23] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC*, March 2004.