Genericity through Dependent Types

Nathaniel Nystrom *
nystrom@us.ibm.com

Igor Peshansky *
igorp@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. Generic types are defined in a natural extension of the dependent type system.

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 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

todo: More positioning, relative to: DML, HM(X), constrained types (Trifonov, Smith) and subtyping constraints, Java generics, GJ, PolyJ, C# generics, virtual types, liquid types

todo: Possible claim: first type system that combines genericity and dep types in some vague general way.

todo: Incorporate some text from OOPSLA paper on deptypes.

todo: Cite liquid types and whatever it cites

X10 is a statically typed object-oriented language designed for high-performance computing [18]. 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, 21, 4, 16, 3, 19, 1, 2, 6, 7, 15], 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 [15], 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 [18, 17]. 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 [18, 17]. 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,

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

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. [17] for a more thorough presentation of constrained types in X10. The following are legal types:

- `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. X10

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.

3. Generics

3.1 Subtyping

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

3.2 Use-site variance

3.3 Class declarations

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} {...}
```

3.4 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]`).

3.5 Class invariant

3.6 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());
```

3.7 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)
        {T <: U}: List[U] = { ... }
   def append[U](other: List[U])
        {T <: U}: List[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`.

3.8 Method overriding

Legal if any call to super method can call sub method.

covariant return contravaraint args weaker where clause

3.9 Constructor definitions

Constructors are defined using the syntax `def this`, as shown in Figure 1. 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.

4. Formal semantics

We present a core calculus, GenX10, for X10 with generics. GenX10 is based on Constrained Featherweight Java [17].

todo: Add method overriding rules: covariant return, contravariant args, weaker constraints

The grammar for GenX10 is shown in Figure 1. The calculus elides features of the full X10 language not relevant to this paper.

4.1 Constraint system

The X10 compiler permits the constraint system to be extended with compiler plugins. The base compiler supports equality constraints over literals and final variables and subtyping and equality constraints over types. The core constraint system is presented here. We assume a constraint solver $\mathcal{C}$ implementing the plugged-in constraint systems.

The constraint system does not distinguish between values and types. Logical variables `x` may represent program variables or type variables. Field accesses and member type references are special cases of atomic formulas; we write `p.f` as sugar for `f(p)`.

The constraint system is shown in Figure 2. $\overline{c}$ is a set of constraints. The constraint system satisfies the structural rules, and supports equality and subtyping constraints over terms.

4.2 Constraint projection

First, for a type environment Γ, we define the *constraint projection*, $\sigma(\Gamma)$ thus:

$$\begin{aligned}\sigma(\varepsilon) &= \texttt{true}\\ \sigma(\Gamma,\texttt{x}:\texttt{T}) &= \sigma(\Gamma), \mathit{cons}(\texttt{T},\texttt{x})\\ \sigma(\Gamma,\texttt{c}) &= \sigma(\Gamma),\texttt{c}\end{aligned}$$

The auxiliary function *cons* specifies the constraint for a type `T` with `self` bounds to `x`. For type variables, the the constraint projection uses an atomic formula `cons`.

$$\begin{aligned}\mathit{cons}(\texttt{C},\texttt{z}) &= \texttt{true}\\ \mathit{cons}(\texttt{C\{c\}},\texttt{z}) &= \texttt{c}[\texttt{z}/\texttt{self}]\\ \mathit{cons}(\texttt{p.X},\texttt{z}) &= \texttt{cons(p.X,z)}\\ \mathit{cons}(\texttt{X},\texttt{z}) &= \texttt{cons(X,z)}\end{aligned}$$

Thus, for example, the constraint projection of the environment:

program	P	::=	$\overline{\texttt{L}}$
classes	L	::=	`class C[`$\overline{\texttt{X}}$`](`$\overline{\texttt{x}}$`:`$\overline{\texttt{T}}$`){c}`
			`  extends T { K `$\overline{\texttt{M}}$` }`
types	T	::=	C
		|	X
		|	e.X
		|	T{c}
		|	$\exists \texttt{x}:\texttt{T}_0.\ \texttt{T}$
		|	$\exists \texttt{X}:\texttt{type}.\ \texttt{T}$
constructors	K	::=	`def this[`$\overline{\texttt{X}}$`](`$\overline{\texttt{x}}$`: `$\overline{\texttt{T}}$`){c}: T = {`
			`  super[`$\texttt{X}_{1..i}$`](`$\texttt{x}_{1..j}$`);`
			`  property[`$\texttt{X}_{i+1..m}$`](`$\texttt{x}_{j+1..n}$`); }`
methods	M	::=	`def m[`$\overline{\texttt{X}}$`](`$\overline{\texttt{x}}$`: `$\overline{\texttt{T}}$`){c}: T = e`
expressions	e	::=	
literals			true | false | null | n
equality		|	$\texttt{e}_1 == \texttt{e}_2$
subtyping		|	$\texttt{T}_1 <: \texttt{T}_2$
type equality		|	$\texttt{T}_1 == \texttt{T}_2$
variables		|	x
field access		|	e.x
val		|	{ val x = $\texttt{e}_0$; $\overline{\texttt{e}}$ }
		|	{ val x: T = $\texttt{e}_0$; $\overline{\texttt{e}}$ }
var		|	{ var x = $\texttt{e}_0$; $\overline{\texttt{e}}$ }
		|	{ var x: T = $\texttt{e}_0$; $\overline{\texttt{e}}$ }
assignment		|	x = e
field assignment		|	$\texttt{e}_1$.x = $\texttt{e}_2$
call		|	$\texttt{e}_0$.m[$\overline{\texttt{T}}$]($\overline{\texttt{e}}$)
		|	$\texttt{e}_0$.m($\overline{\texttt{e}}$)
new		|	new C[$\overline{\texttt{T}}$]($\overline{\texttt{e}}$)
		|	new C($\overline{\texttt{e}}$)
cast		|	e as T
if		|	if ($\texttt{e}_0$) $\texttt{e}_1$ else $\texttt{e}_2$
constraint terms	t	::=	
literals			true | n | C
variables		|	x | X
properties		|	t.x | t.X
atoms		|	$\texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_n)$
constraint	c	::=	true
		|	$\texttt{t}_1 == \texttt{t}_2$
		|	$\texttt{t}_1 <: \texttt{t}_2$
		|	$\exists \texttt{x}.\ \texttt{c}$
		|	$\overline{\texttt{c}}$
		|	$\texttt{p}(\texttt{t}_1, \ldots, \texttt{t}_n)$
environments	Γ	::=	ε
		|	$\Gamma, \texttt{c}$
		|	$\Gamma, \texttt{x}:\texttt{T}$
		|	$\Gamma, \texttt{X}:\texttt{type}$

Figure 1. GenX10 grammar

$$\frac{}{\overline{\texttt{c}} \Vdash \texttt{c}_i} \quad \text{(I)}$$

$$\frac{\overline{\texttt{c}} \Vdash_{\mathcal{X}} \texttt{d}}{\overline{\texttt{c}} \Vdash \texttt{d}} \quad \text{(PLUGIN)}$$

$$\frac{\overline{\texttt{c}} \Vdash \texttt{d}[\texttt{t}/\texttt{x}]}{\overline{\texttt{c}} \Vdash \exists \texttt{x}.\ \texttt{d}} \quad \text{(EX-I)}$$

$$\frac{\overline{\texttt{c}}, \texttt{d}[\texttt{t}/\texttt{x}] \Vdash \texttt{e} \qquad \overline{\texttt{c}} \Vdash \exists \texttt{x}.\ \texttt{d} \qquad \texttt{x} \notin FV(\texttt{e})}{\overline{\texttt{c}} \Vdash \texttt{e}} \quad \text{(EX-E)}$$

$$\frac{\overline{\texttt{c}} \Vdash \overline{\texttt{s}} == \overline{\texttt{t}}}{\overline{\texttt{c}} \Vdash \texttt{f}(\overline{\texttt{s}}) == \texttt{f}(\overline{\texttt{t}})} \quad \text{(EQ-ATOM)}$$

$$\frac{}{\overline{\texttt{c}} \Vdash \texttt{t} == \texttt{t}} \quad \text{(EQ-REFL)}$$

$$\frac{\overline{\texttt{c}} \Vdash \texttt{t}_1 == \texttt{t}_2 \qquad \overline{\texttt{c}} \Vdash \texttt{t}_2 == \texttt{t}_3}{\overline{\texttt{c}} \Vdash \texttt{t}_1 == \texttt{t}_3} \quad \text{(EQ-TRANS)}$$

$$\frac{\overline{\texttt{c}} \Vdash \texttt{t}_1 == \texttt{t}_2}{\overline{\texttt{c}} \Vdash \texttt{t}_2 == \texttt{t}_1} \quad \text{(EQ-SYM)}$$

$$\frac{\overline{\texttt{c}} \Vdash \texttt{T}_1 <: \texttt{T}_2 \quad \overline{\texttt{c}} \Vdash \texttt{T}_2 <: \texttt{T}_1}{\overline{\texttt{c}} \Vdash \texttt{T}_1 == \texttt{T}_2} \quad \text{(EQ-SUB)}$$

$$\frac{\overline{\texttt{c}} \Vdash \texttt{C\{c\}}:\texttt{type} \qquad \overline{\texttt{c}}, \texttt{c} \Vdash \texttt{d}}{\overline{\texttt{c}} \Vdash \texttt{C\{c\}} <: \texttt{C\{d\}}} \quad \text{(SUB-CONS)}$$

$$\frac{\texttt{C}[\overline{\texttt{X}}](\overline{\texttt{x}}:\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{ext T \{ K}\overline{\texttt{M}}\ \overline{\texttt{F}}\ \texttt{\}}}{\Vdash \texttt{C} <: \texttt{T}} \quad \text{(SUB-SUPER)}$$

$$\Vdash \texttt{T} <: \texttt{Object} \quad \text{(SUB-OBJECT)}$$

$$\Vdash \texttt{Null} <: \texttt{T} \quad \text{(SUB-NULL)}$$

$$\frac{\overline{\texttt{c}} \Vdash \texttt{T}_1 == \texttt{T}_2}{\overline{\texttt{c}} \Vdash \texttt{T}_1 <: \texttt{T}_2} \quad \text{(SUB-EQ)}$$

$$\frac{\overline{\texttt{c}} \Vdash \texttt{T}_1 <: \texttt{T}_2 \qquad \overline{\texttt{c}} \Vdash \texttt{T}_2 <: \texttt{T}_3}{\overline{\texttt{c}} \Vdash \texttt{T}_1 <: \texttt{T}_3} \quad \text{(SUB-TRANS)}$$

Figure 2. Constraints

$$\frac{\texttt{C[}\overline{\texttt{X}}\texttt{](}\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{)\{c\} ext T \{ }\overline{\texttt{M}}\ \overline{\texttt{F}}\texttt{ \}}}{\vdash \texttt{C:type}}$$

$$\frac{\Gamma \vdash \texttt{T:type} \qquad \Gamma, \texttt{self:T} \vdash \texttt{c:Boolean} \qquad \sigma(\Gamma) \Vdash \texttt{c OK}}{\Gamma \vdash \texttt{T\{c\}:type}}$$

$$\frac{\Gamma \vdash \texttt{p:T} \qquad \Gamma \vdash \texttt{T has X}}{\Gamma \vdash \texttt{p.X:type}}$$

$$\Gamma, \texttt{X:type} \vdash \texttt{X:type}$$

Figure 3. Type well-formedness

$$\frac{\texttt{C[}\overline{\texttt{X}}\texttt{](}\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{)\{c\} ext T \{ K }\ \overline{\texttt{M}}\ \overline{\texttt{F}}\texttt{ \}}}{\begin{array}{c}\vdash \texttt{C has K} \\ \vdash \texttt{C has } \texttt{X}_i \\ \vdash \texttt{C has } \texttt{x}_i\texttt{:T}_i \\ \vdash \texttt{C has } \texttt{M}_i \\ \vdash \texttt{C has } \texttt{F}_i\end{array}} \text{ (HAS-CLASS)}$$

$$\frac{\texttt{Z} \neq \texttt{K} \qquad \Gamma \vdash \texttt{T}_1 \texttt{ has Z} \qquad \sigma(\Gamma) \vdash \texttt{T}_2 \texttt{<:} \texttt{T}_1}{\Gamma \vdash \texttt{T}_2 \texttt{ has Z}} \text{ (HAS-SUB)}$$

Figure 4. Structural constraints

```
b: D, a: C{self.X==D{d},self.Y<:b.Z}
```

is:

```
a.X==D{d}, a.Y<:b.Z
```

4.3 Type well-formedness

4.4 Type inference rules

4.4.1 Constraint rules

4.4.2 Expression typing judgment

The cast rule T-CAST requires that the cast type be well-formed.

The field access rule T-FIELD differs from the rule in the paper in that there is no need to substitute a fresh variable for the receiver. Note that `this` may be free in S—that would be a reference to the current object in the code in which `e.f` occurs, not a reference to the receiver of the `e.f` field selection (i.e., the object obtained by evaluating `e`).

if we allow adding constraints to arbitrary types—do we?

TODO: type parameters!

Now we consider the rule for method invocation. Assume that in a type environment Γ the expressions $\texttt{e}_\texttt{0}, \ldots, \texttt{e}_\texttt{n}$ have the types $\texttt{T}_\texttt{0}, \ldots, \texttt{T}_\texttt{n}$. Since the actual values of these expressions are not known, we shall assume that they take on some fixed but unknown values $\texttt{z}_\texttt{0}, \ldots, \texttt{z}_\texttt{n}$ of types $\texttt{T}_\texttt{0}, \ldots, \texttt{T}_\texttt{n}$. Now, for $\texttt{z}_\texttt{0}$ as receiver, let us assume that the type $\texttt{T}_\texttt{0}$ has a method named `m` with signature $[\overline{\texttt{Z}}](\overline{\texttt{z}}\texttt{:}\overline{\texttt{S}})\texttt{\{c\}} \to \texttt{U}$ (Let $\texttt{T}_\texttt{0} = \texttt{C\{d\}}$. If there is no method named `m` for the class `C` then this method invocation cannot be type-checked. Without loss of generality, we may assume that the type parameters of this method are named $\texttt{Z}_1, \ldots, \texttt{Z}_\texttt{k}$, and the value parameters are named $\texttt{z}_1, \ldots, \texttt{z}_\texttt{n}$ since we are free to choose variable names as we wish.) Now, for the method to be invocable, it must be the case that the types $\texttt{T}_1, \ldots, \texttt{T}_\texttt{n}$ are subtypes of $\texttt{S}_1, \ldots, \texttt{S}_\texttt{n}$. (Note that there may be no occurrences of `this` in $\texttt{S}_1, \ldots, \texttt{S}_\texttt{n}$—they have been replaced

$$\frac{\Gamma \vdash \texttt{e:S} \qquad \sigma(\Gamma) \Vdash \texttt{S<:T} \qquad \Gamma \vdash \texttt{T:type}}{\Gamma \vdash \texttt{e:T}} \text{ (T-SUB)}$$

$$\begin{array}{c}\vdash \texttt{true:Boolean\{self==true\}} \\ \vdash \texttt{false:Boolean\{self==false\}}\end{array} \text{ (T-BOOL)}$$

$$\vdash n\texttt{:Int\{self==}n\texttt{\}} \text{ (T-INT)}$$

$$\vdash \texttt{null:Null} \text{ (T-NULL)}$$

$$\frac{\Gamma \vdash \texttt{e}_1\texttt{:T}_1 \qquad \Gamma \vdash \texttt{e}_2\texttt{:T}_2}{\Gamma \vdash \texttt{e}_1\texttt{==e}_2\texttt{:}\exists \texttt{z}_1\texttt{:T}_1, \texttt{z}_2\texttt{:T}_2.\ \texttt{Boolean\{self==(z}_1\texttt{==z}_2\texttt{)\}}} \text{ (T-EQ)}$$

$$\frac{\Gamma \vdash \texttt{T}_1\texttt{:type} \qquad \Gamma \vdash \texttt{T}_2\texttt{:type}}{\Gamma \vdash \texttt{T}_1\texttt{==T}_2\texttt{:Boolean}} \text{ (T-TEQ)}$$

$$\frac{\Gamma \vdash \texttt{T}_1\texttt{:type} \qquad \Gamma \vdash \texttt{T}_2\texttt{:type}}{\Gamma \vdash \texttt{T}_1\texttt{<:T}_2\texttt{:Boolean}} \text{ (T-TSUB)}$$

$$\Gamma, \texttt{x:T} \vdash \texttt{x:T} \text{ (T-VAR)}$$

TODO: Tself==x

$$\frac{\Gamma \vdash \texttt{e}_0\texttt{:T}_0 \qquad \Gamma, \texttt{x:T}_0 \vdash \overline{\texttt{e}}\texttt{:}\overline{\texttt{T}}}{\Gamma \vdash \texttt{\{ val x = e}_0\texttt{; }\overline{\texttt{e}}\texttt{ \}:T}_n} \text{ (T-VALDEC)}$$

$$\frac{\Gamma \vdash \texttt{e}_0\texttt{:T}_0 \qquad \Gamma, \texttt{x:T}_0 \vdash \overline{\texttt{e}}\texttt{:}\overline{\texttt{T}}}{\Gamma \vdash \texttt{\{ val x:T}_0\texttt{ = e}_0\texttt{; }\overline{\texttt{e}}\texttt{ \}:T}_n} \text{ (T-VALDEC2)}$$

$$\frac{\Gamma \vdash \texttt{e:S} \qquad \Gamma \vdash \texttt{T:type}}{\Gamma \vdash \texttt{e as T:T}} \text{ (T-CAST)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e:T} \\ \texttt{T has f\{c\}:U} \\ \sigma(\Gamma, \texttt{this:T}) \Vdash \texttt{c}\end{array}}{\Gamma \vdash \texttt{e.f:}\exists \texttt{this:T.\ U}} \text{ (T-FIELD)}$$

TODO: Uself==this.f

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0\texttt{:T}_0 \qquad \Gamma \vdash \overline{\texttt{e}}\texttt{:}\overline{\texttt{T}} \\ \texttt{T}_0 \texttt{ has def m[}\overline{\texttt{X}}\texttt{](}\overline{\texttt{x}}\texttt{:}\overline{\texttt{S}}\texttt{)\{c\}:U = e} \\ \Gamma' = \Gamma, \overline{\texttt{X}}\texttt{:type}, \texttt{this:T}_0, \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}, \overline{\texttt{X}}\texttt{==}\overline{\texttt{V}} \\ \sigma(\Gamma') \Vdash \texttt{c} \\ \sigma(\Gamma') \Vdash \overline{\texttt{T}}\texttt{<:}\overline{\texttt{S}}\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.m[}\overline{\texttt{V}}\texttt{](}\overline{\texttt{e}}\texttt{):}\exists \overline{\texttt{X}}\texttt{:type}, \texttt{this:T}_0, \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{.\ U}} \text{ (T-INVK)}$$

$$\frac{\begin{array}{c}\overline{\texttt{Y}} \text{ fresh} \\ \Gamma, \overline{\texttt{Y}}\texttt{:type} \vdash \texttt{e}_0\texttt{.m[}\overline{\texttt{Y}}\texttt{](}\overline{\texttt{e}}\texttt{):T}\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{):T}} \text{ (T-INVK-INFERRED)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{\texttt{e}}\texttt{:}\overline{\texttt{T}} \\ \texttt{C has def this[}\overline{\texttt{X}}\texttt{](}\overline{\texttt{x}}\texttt{:}\overline{\texttt{S}}\texttt{)\{c\}:U = } \ldots \\ \Gamma' = \Gamma, \overline{\texttt{X}}\texttt{:type}, \texttt{this:C}, \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}, \overline{\texttt{V}}\texttt{==}\overline{\texttt{X}} \\ \Gamma'' = \Gamma, \overline{\texttt{X}}\texttt{:type}, \texttt{this:U}, \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}, \overline{\texttt{V}}\texttt{==}\overline{\texttt{X}} \\ \sigma(\Gamma') \Vdash \texttt{c} \\ \sigma(\Gamma') \Vdash \overline{\texttt{T}}\texttt{<:}\overline{\texttt{S}} \\ \sigma(\Gamma'') \Vdash \mathit{inv}(\texttt{C}),\end{array}}{\Gamma \vdash \texttt{new C[}\overline{\texttt{V}}\texttt{](}\overline{\texttt{e}}\texttt{):}\exists \overline{\texttt{X}}\texttt{:type}, \texttt{this:C}, \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{.\ U}} \text{ (T-NEW)}$$

$$\frac{\begin{array}{c}\overline{\texttt{Y}} \text{ fresh} \\ \Gamma, \overline{\texttt{Y}}\texttt{:type} \vdash \texttt{new C[}\overline{\texttt{Y}}\texttt{](}\overline{\texttt{e}}\texttt{):T}\end{array}}{\Gamma \vdash \texttt{new C(}\overline{\texttt{e}}\texttt{):T}} \text{ (T-NEW-INFERRED)}$$

by $\mathtt{z_0}$.) Further, it must be the case that for these parameter values, the constraint $\mathtt{c}$ is entailed. Given all these assumptions it must be the case that the return type is $\mathtt{U}$, with all the parameters $\mathtt{z_0}, \ldots, \mathtt{z_n}$ existentially quantified.

4.4.3 Class OK judgment

The following rule is modified from what we had in the paper to ensure that all the types are well-formed (under the assumption $\mathtt{this}:\mathtt{C}$). Note that the variables $\overline{\mathtt{x}}$ are permitted to occur in the types $\mathtt{T}_0, \overline{\mathtt{T}}$, hence their typing assertions must be added to Γ.

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

This rule did not exist in our submission. This is necessary to ensure that the types of fields are well-formed.

$$\frac{\mathtt{this}:\mathtt{C}, \mathtt{c} \vdash \mathtt{T}:\mathtt{type}}{\mathtt{val}\ \mathtt{f}\{\mathtt{c}\}:\mathtt{T}\ \text{OK in}\ \mathtt{C}} \quad (\textsc{Field OK})$$

This rule is now modified to ensure that all the types and methods in the body of the class are well-formed.

$$\frac{\begin{array}{c}K\ \text{OK in}\ \mathtt{C} \\ \overline{\mathtt{M}}\ \text{OK in}\ \mathtt{C} \\ \overline{\mathtt{F}}\ \text{OK in}\ \mathtt{C} \\ \mathtt{this}:\mathtt{C} \vdash \mathtt{T}:\mathtt{type}\end{array}}{\mathtt{C}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{ext}\ \mathtt{T}\ \{\ K\ \overline{\mathtt{M}}\ \overline{\mathtt{F}}\ \}\ \text{OK}} \quad (\textsc{Class OK})$$

TODO: method overriding

4.4.4 Subtype judgment

$$\frac{\sigma(\Gamma) \vdash_{\mathcal{C}} \mathtt{T}_1 <: \mathtt{T}_2}{\Gamma \vdash \mathtt{T}_1 <: \mathtt{T}_2}$$

5. Odds and ends

static methods cannot mention T

interfaces can have static methods; a property can implement I, allowing T.m() static calls

Extensions: structural constraints, optional methods, interfaces enabled flag trick

6. Constraint solver

The goal of the constraint solver is to check an assertion $\mathtt{c} \Vdash \mathtt{d}$.

The first step is to normalize constraints. Normalization splits constraints into type constraints and value constraints. Produces a set of constraint judgments $\overline{\mathtt{c}} \Vdash \mathtt{c}$ where $\mathtt{c}$ contains no conjunctions.

$$\begin{aligned}[\![\overline{\mathtt{c}} \Vdash \mathtt{d}_1, \mathtt{d}_2]\!] &= \overline{\mathtt{c}} \Vdash \mathtt{d}_1, \overline{\mathtt{c}} \Vdash \mathtt{d}_2 \\ [\![\overline{\mathtt{c}} \Vdash \mathtt{T}_1 <: \mathtt{T}_2]\!] &= \overline{\mathtt{c}} \Vdash \mathit{class}(\mathtt{T}_1) <: \mathit{class}(\mathtt{T}_2), \\ &\quad\ \overline{\mathtt{c}}, \mathit{cons}(\mathtt{T}_1, \mathtt{self}) \Vdash \mathit{class}(\mathtt{T}_2) \\ [\![\overline{\mathtt{c}} \Vdash \mathtt{d}]\!] &= \overline{\mathtt{c}} \Vdash \mathtt{d} \qquad \text{(otherwise)}\end{aligned}$$

Once in normalized form, the inference proceeds as follows: Select a constraint $\overline{\mathtt{c}} \Vdash \mathtt{c}$. If not consistent, fail. If valid, ok. If not valid, generate assignment of variables that makes it true, adding the assignment to the assumptions for all constraints.

The inference algorithm must specify the criteria for:

- selecting the next constraint to solve
- generating the variable assignment consistent with all other constraints (to avoid backtracking)

Pick an unassigned variable, find weakest assignment that makes just this clause true. Does the weakest assignment exist?

Question: can we ensure each clause involves only one or two unknowns?

6.1 A graph-based inference algorithm

Represent a constraint as a graph G. Each node represents a constraint term for a value or a type. The node for a path p is written v_p; the node for a type T is written V_T. There are four kinds of edges:

1. undirected equivalence edges, $v_p \sim v_q$ and $V_S \sim V_T$,
2. type edges, $v_p \mapsto_{\mathtt{type}} V_T$,
3. tree edges, $v_p \mapsto_f v_{p.f}$ and $v_p \mapsto_X V_{p.X}$, and
4. flow edges, $V_S \to V_T$.

First, each constraint term is mapped to a node in the graph as follows. Associate each term t with a node v_t. For each access path $\mathtt{p.x}$, add a tree edge $v_{\mathtt{p}} \mapsto_{\mathtt{x}} v_{\mathtt{p.x}}$. For each path type $\mathtt{p.X}$, add a tree edge $v_{\mathtt{p}} \mapsto_{\mathtt{X}} V_{\mathtt{p.X}}$. For each atomic formula $\mathtt{f}(\overline{\mathtt{t}})$, add the tree edge $v_{\mathtt{f}(\overline{\mathtt{t}})} \mapsto_i v_{t_i}$ for all i. If term t has type T, add $v_t \mapsto_{\mathtt{type}} V_{t.\mathtt{type}}$ and add $V_T \sim V_{t.\mathtt{type}}$ to G.

Type nodes are sets of classes.

Next, constraints are incorporated into the graph:

- For constraint $\mathtt{p}==\mathtt{q}$, add $v_{\mathtt{p}} \sim v_{\mathtt{q}}$ to G.
- For constraint $\mathtt{S}==\mathtt{T}$, add $V_{\mathtt{S}} \sim V_{\mathtt{T}}$ to G.
- For constraint $\mathtt{S}<:\mathtt{T}$, add $V_{\mathtt{S}} \to V_{\mathtt{T}}$ to G.

A flow-path is a path that follows flow and equivalence edges only. A type-path is a path that follows type and equivalence edges only.

Now, we saturate: If there is a type-path $v_t \mapsto^*_{\mathtt{type}} V_{\mathtt{C}\{\mathtt{c}\}}$, add $c[t/\mathtt{self}]$ to the worklist.

Can saturate lazily when doing a lookup. EXCEPT: a type may have an arbitrary constraint `C{self.x==3 && y > 7}` EXCEPT: c is x.f==... with x: Cc need to avoid infinite loop

The inference algorithm adds flow edges, $v \to w$, to the graph G.

For constraint $\mathtt{S}<:\mathtt{T}$, add the flow edge $v_{\mathtt{S}} \to v_{\mathtt{T}}$ to G.

TODO: self constraints TODO: existentials

Initialize W to the set of equivalence edges for the set of equational constraints $\mathtt{p}==\mathtt{q}$. Initialize G with flow edges only.

Merge fields. For all t in the constraint, for all $v_{t'}$ reachable by flow edges from v_t, if t has a type member $\mathtt{X}$, add $v_{t.\mathtt{X}} \sim v_{t'.\mathtt{X}}$ to W; if t has a field $\mathtt{f}$, add $v_{t.\mathtt{f}.\mathtt{type}} \sim v_{t'.\mathtt{f}.\mathtt{type}}$ to W.

Process equational constraints. While W is non-empty, extract and remove $v_{\mathtt{p}} \sim v_{\mathtt{q}}$ from W. If the edge is already in G, continue with the next constraint.

If the edge connects $v_{\mathtt{S}}$ and $v_{\mathtt{T}}$ and $\mathtt{S}$ and $\mathtt{T}$ cannot be equal, fail.

If $v_{p.X} \sim v_T$ and there is a type edge from v_p to $v_{\mathtt{C}\{\mathtt{c}\}}$ add in $\mathtt{c}[\mathtt{p}/\mathtt{self}]$.

If adding the edge would create a cycle, merge the nodes.

Let $\mathit{src}(v)$ be the set of class/interface nodes from which v is reachable by flow and equivalence edges. These are the nodes that represent subtypes of v's type.

Let $\mathit{snk}(v)$ be the set of class/interface nodes reachable from v by flow and equivalence edges. These are the nodes that represent supertypes of v's type.

If there is a v in both $src(v_p)$ and $snk(v_q)$, add $v \sim v_p$ to W.
If there is a v in both $src(v_q)$ and $snk(v_p)$, add $v \sim v_q$ to W.
Add the edge to G and continue with W.

7. 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.

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

7.1 Discussion

Consider the following method from [21]:

```
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 [21], 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];
```

8. Implementation

This section describes an implementation approach for constrained types on a Java virtual machine. We describe the implementation as a translation to Java.

The design is a hybrid design based on the implementation of parameterized classes in NextGen [1, 2] and the implementation of PolyJ [3]. Generic classes are translated into template classes that are instantiated on demand at run time by binding the type properties to concrete types. To implement run-time type checking (e.g., casts), type properties are represented at run time using *adapter objects*.

This design, appropriately extended to handle language features not described in this paper, has been implemented in the X10 compiler. The X10 compiler is built on the Polyglot framework and translates X10 source to Java source[1]

8.1 Method parameters

The first step in translation is to remove method parameters by introducing a generic member class for each generic method. The member class is static iff the method is static. Constructor type parameters are left unchanged. After this step, the code consists only of generic classes. The remaining translation introduces a run-time representation for the type properties of these classes.

[1] There is also a translation from X10 to C++ source, not described here.

8.2 Classes

Each class is translated into a *template class*. The template class is compiled by a Java compiler (e.g., javac) to produce a class file. At run time, when a constrained type `C{c}` is first referenced, a class loader loads the template class for `C` and then transforms the template class bytecode, specializing it to the constraint `c`.

For example, consider the following classes.

```
class A[T] {
    var a: T;
}
class C {
    val x: A[Int] = new A[Int]();
    val y: Int = x.a;
}
```

The compiler generates the following code:

```
class A {
    // Dummy class needed to type-check uses of T.
    @TypeProperty(1) static class T { }

    T a;

    // Dummy getter and setter; will be eliminated
    // at run time and replaced with actual gets
    // and sets of the field a.
    @Getter("a") <S> S get$a() { return null; }
    @Setter("a") <S> S set$a(S v) { return null; }
}

class C {
    @ActualType("A$Int")
    final A x = Runtime.<A>alloc("A$Int");
    final int y = x.<Integer>get$a();
}
```

The member class `A.T` is used in place of the type property `T`. The `Runtime.alloc` method is used used in place of a constructor call. This code is compiled to Java bytecode.

Then, at run time, suppose the expression `new C()` is evaluated. This causes `C` to be loaded. The class loader transforms the bytecode as if it had been written as follows:

```
class C {
    final A$Int x = new A$Int();
    final int y = x.a;
}
```

The `ActualType` annotation is used to change the type of the field `x` from `A` to `A$Int`. The call to `Runtime.alloc` is replaced with a constructor call. The call to `x.get$a()` is replaced with a field access.

The implementation cannot generate this code directly because the class `A$Int` does not yet exist; the Java source compiler would fail to compile `C`.

Next, as the `C` object is being constructed, the expression `new A$Int()` is evaluated, causing the class `A$Int` to be loaded. The class loader intercepts this, demangles the name, and loads the bytecode for the template class `A`.

The bytecode is transformed, replacing the type property `T` with the concrete type `int`, the translation of `Int`.

```
class A {
    x10.runtime.Type T;
}

class A$Int extends A {
    int x;
}
```

expressions	e	::=	...
		|	T has Sig
signatures	Sig	::=	def this[$\overline{X}$]($\overline{x}$: $\overline{T}$){c}: T
		|	def m[$\overline{X}$]($\overline{x}$: $\overline{T}$){c}: T
		|	val x{c}: T

Figure 10. Grammar for structural constraints

Type properties are mapped to the Java primitive types and to Object. Only nine possible instantiations per parameter. Instantiations used for representation. Adapter objects used for run time type information.

Could do instantiation eagerly, but gets out of hand quickly: 9 instantiations for one type properties, 81 for two type properties, 729 for three. Most of these are not used.

Value constraints are erased from type references.

Constructors are translated to static methods of their enclosing class. Constructor calls are translated to calls to static methods.

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

8.3 Eliminating method type parameters

8.4 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 name of the actual type property.[2] 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.

9. Structural constraints

XXX this is an extension of the type system

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 10.

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

```
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

Structural constraints on types are found in many languages. Haskell [10] 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)`$[$`T`/`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`

```
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

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) ... }
}
```

```
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

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

10. Discussion

10.1 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.

10.2 Wildcards

Wildcards in Java [9, 21] were motivated by the following example (rewritten in X10 syntax) from [21]. 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] = { ... }
}
```

10.3 Proper abstraction

Consider the following example adapted from [21]:

```
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.

10.4 Virtual types

Type properties share many similarities with virtual types [12, 11], particularly with sound formulations of virtual types using path-dependent types, as found in gbeta [8], Scala [15], and J& [14]. 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 [19] 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 = { ... }
}
```

`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}`.

11. Related work

[20] [21] [7] [13] [3] [2] [1] [11] [12] [19]

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.

13. Acknowledgements

The authors thank Doug Lea, Lex Spoon, Jens Palsberg, Bob Blainey, and Olivier Tardieu for valuable feedback on versions of the language. The implementation uses some code from the implementation of PolyJ by Michael Clarkson and Andrew Myers.

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.

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

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

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

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

[14] 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.

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

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

[17] 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.

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

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

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

[21] 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.