Genericity through Constrained Types

Nathaniel Nystrom Olivier Tardieu Igor Peshansky Vijay Saraswat
IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights NY 10598 USA
{nystrom,tardieu,igorp,vsaraswa}@us.ibm.com

Abstract

Modern object-oriented languages such as X10 require a rich framework for types capable of expressing value-dependency, type-dependency and supporting pluggable, application-specific extensions.

In earlier work, we presented the framework of *constrained types* for concurrent, object-oriented languages, parametrized by an underlying constraint system $\mathcal{C}$. Constraint systems are a very expressive framework for partial information. Types are viewed as formulas `C{c}` where `C` is the name of a class or an interface and `c` is a constraint in $\mathcal{C}$ on the immutable instance state of `C` (the *properties*). Many (value-)dependent type systems for object-oriented languages can be viewed as constrained types.

This paper extends the constrained types approach to handle *type-dependency* ("genericity"). The key idea is to extend the constraint system to include predicates over types such as `X` is a subtype of `T`. Generic types are supported by introducing parameters and properties that range over types and permitting the user program to impose constraints on such variables. Type-valued properties are required to have a run-time representation—the run-time semantics is not defined through erasure. Run-time casts are permitted through dynamic code generation.

To illustrate the underlying theory, we develop a formal family FX($\mathcal{C}$) of programming languages with a common set of sound type-checking rules. By varying $\mathcal{C}$ and extending the type system, we obtain languages with the power of FJ, FGJ, dependent types, and new object-oriented languages that uniformly support value- and type-dependency. Concretely, we illustrate with the design and implementation of the type system for X10. The type system integrates and extends the features of nominal types, virtual types, and path-dependent types.

1. Introduction

Modern architectural advances are leading to the development of complex computational systems, such as heterogeneous multi-core, hardware accelerators, and large CPU-count hybrid clusters. The X10 programming language [47, 7, 46] was designed to address the challenge of developing high-performance applications for such machines, building on the productivity gains of modern object-oriented languages.

X10 requires a rich type system to enable code reuse, rule out a large variety of errors at compile-time, and to generate efficient code. A central data structure in X10 is the dense, distributed, multi-dimensional array. Arrays are defined over a set of indices known as *regions*, and support arbitrary base types and accesses through *points* that must lie in the underlying region. For performance it is necessary that array accesses are bounds-checked statically as far as possible. Further, certain regions (such as polyhedral regions) may be represented particularly efficiently. Hence, if a variable is to range only over polyhedral regions, it is important that this information be conveyed statically (through the type system) to the code generator. To support *P*-way data parallelism it is often necessary to logically partition an array into *P* pieces. A type system that can establish that a given division is a partition can ensure that no race conditions arise due to simultaneous accesses by multiple activities to different pieces.

1.1 Constrained types

These requirements motivated us to develop a framework for dependent types in object-oriented languages [40]. *Dependent type systems* [30, 55, 10] have been extensively developed over the past few decades in the context of logic and functional programming—they permit types to be parametrized by *values*.

The key idea behind our approach is to focus on the notion of a *constraint system* [45]. Constraint systems were originally developed to provide a simple framework for a large variety of inference systems used in programming languages, in particular as a foundation for constraint programming languages. Patterned after Scott's information systems, a constraint system is organized around the notion of *constraints* or tokens of partial information (e.g., `X+Y>Z*3`), together with an entailment relation $\vdash$. Tokens may have first-order structure; existential quantification is supported. The entailment relation is required to support a certain set of inference rules arising from a Gentzen-style formulation of intuitionistic logic.

In applying constraint systems to object-oriented languages[1], the principal insight is that objects typically have some immutable state, and constraints on this state are of interest to the application. For instance, in Java the length of an array might not be statically known, but is fixed once the array is created. Hence, we can enrich the notion of a type: for a class `C` we permit a *constrained type* `C{c}` where `c` is a constraint on the immutable fields, or *properties*, of the class [40]. Thus, `Array{self.length==N}` is a type satisfied by any array whose length is `N`—a (final) variable whose value may be unknown statically. In a constraint, `self` refers to a value of the base type being constrained, in this case `Array`. Subtyping is easily defined: a type `C{c}` is a subtype of `D{d}` provided that `C` is a subclass of `D` and `c` entails `d` in the underlying constraint system.

Constrained types maintain a phase distinction between compile time (entailment checking in the underlying constraint system) and run time (computation). Dynamic type casting is permitted—code is generated to check at run time that the properties of the given object satisfy the given constraint.

The constrained types approach enjoys many nice properties in contrast to similar approaches such as DML [55]. Constrained types are a natural extension to OO languages, and quite easy to use. Constraints may also be used to specify class invariants, and conditions on the accessibility of fields and methods (conditional fields and methods). Final variables in the computation can be used directly in types; there is no need to define a separate parallel language of index expressions to be used in the type system. Constrained types always permit field selection and equality at object types; hence the programmer may specify constraints at any user-specified object type, not just over the built-in constraint system.

1.2 Generic types

In this paper we extend the constrained types approach to handle *generic types* [26, 37, 4, 35, 50, 18, 49]—types such as `List<T>` in Java that are parametrized by other types. Generic types are vital for implementing type-safe, reusable libraries, especially collections classes. For instance, the data type `Array` discussed above is generic on its member type.

To permit genericity, variables `X` must be admitted over types. What constraints can be used to specify conditions on such variables? In nominally typed OO languages such as Java, the answer is relatively clear and a simple semantic framework can be sketched out: a type is a class *name*.[2] Intuitively, an object belongs to a type if it is an instance of the class. Types are equipped with a partial order (the *subtyping* order) generated from the user program through the "`extends`" relationship. Further, each type is associated with member fields and methods, each with their name and signature.

This motivates a very natural constraint system on types. For a type variable `X` we should be able to assert the constraint `X <= T`: a valuation (mapping from variables to types) realizes this constraint if it maps `X` to a type that extends `T`. Similarly, it should be able to require that a type has a particular member—a field with a given name and type, or a method with a given name and signature. We introduce the constraints `T has f:T` and `T has` $\mathtt{m}(\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}})\mathtt{:T}$ to express this.

The next question is: where should type variables be permitted? Clearly, one must permit methods to have type parameters (as permitted for instance in Java [18], Scala [41], and in functional languages such as ML [33] and Haskell [23]). It is necessary as well to permit classes to be dependent on types—for instance, `Array` should be dependent on the type of its member elements. While it is possible to develop an approach in which classes have type *parameters* [18], the constrained type approach suggests an alternative: classes may have type-valued properties. To create an instance of such a class, a type must be supplied to initialize this property. As with other object state, this value is available at runtime and can hence be used in dynamic cast expressions.

A pleasing aspect of the resulting design is that the same fundamental mechanism of constrained types—imposing constraints on properties—is used to specify both value-dependency and type-dependency of types. Consider the class `Array` declared as:

```
class Array[T](r: Region) {
  def get(p: Point{self in r}): T = ...;
  def set(p: Point{self in r}, v: T) = ...;
  ...
}
```

The class has two properties: a type-valued property `T` and a value property `r` of type `Region`. The `get` method for the array requires a point `p` that must lie in the object's region and returns a value of type `T`. Within the body of a class, class members may be referenced without using the "`this.`" selector, as usual for OO languages. Hence occurrences of `this.r` and `this.T` can be written `r` and `T`, respectively. Similarly, the `set` method takes a point in the region and a value of type `T`.

The constrained type `Array{self.T==int}` specifies the type of all arrays whose element type (`T`) is `int`. `Array{T<=Number, r.rank==2}` specifies the type of all arrays whose base type is a subtype of `Number`, and whose region has rank 2. The type `Array{T==List{length==N}, r.rank==N}` specifies an `N`-dimensional array whose elements are lists (of unknown type) that are precisely `N` long.

Note that for any expression `a` of type `Array`, `a.T` is a type, equivalent to the type to which `T` was initialized when the object `a` was instantiated (cf. the return type `this.T` for

[1] The use of constraints for types has a distinguished history going back to Mitchell [34]. Our work is closely related to the HM(X) approach [48]—see Section 6 for details.

[2] This can be extended naturally to account for interfaces.

`Array.get`). For soundness, we require that `a` be a final access path: an immutable variable (including `this`) or an access to a final field of a final access path.

1.3 Design and implementation of the X10 type system

Many features of modern object-oriented type systems fall out naturally in this extended framework for constrained types. We illustrate these by discussing the design of the X10 language and of various programming idioms.

This framework forms the basis of the X10 type system. To make the system more usable, the X10 language design simplifies the type system by restricting how constraints over type properties can be used. We discuss these restrictions in Section 5.

Since there may be a large number of constrained types in a program, a pure heterogeneous translation in which a class is instantiated multiple times on different constraints may lead to significant code bloat. Instead we use a hybrid implementation scheme that combines ideas from NextGen [6, 1] and PolyJ [35]. The implementation supports run-time type introspection and instantiation of generic types on primitive types. The performance of primitive arrays, especially, is critical for the high-performance applications for which X10 is intended. Our design does not require that primitive values be boxed.

1.4 The FX($\mathcal{C}$) family

To further the foundations of constrained types, we develop the FX family of languages. The core expression language is essentially Featherweight Java (FJ [21]) with constrained types over value and type constraints. A single set of rules specifies the static and dynamic semantics for all languages in the family. The static semantics is shown to be sound with respect to the operational semantics.

Different members of the family are associated with different constraint systems. FX($\cdot$) is FX instantiated over the vacuous constraint system—the only user-specifiable constraint permitted is the vacuous `true`. FX($\cdot$) corresponds to FJ. FX($\mathcal{G}$) permits user-specifiable subtyping and type equality constraints; it corresponds to FGJ (being somewhat richer in permitting path types). FX($\mathcal{A}$) permits the use of constraints from some constraint system $\mathcal{A}$ over values; it corresponds to a language with a pure value-dependent type system as described in [40]. Finally, FX($\mathcal{G}$,$\mathcal{A}$) permits the use of both value and type constraints, and is the main topic of this paper.

Contributions. We extend the constrained types approach to handle generic types. We present the design and implementation of the type system for a concrete language X10 based on these ideas. We show how several other ideas in OO typing (such as structural types) can also be handled in this framework. We present a family of formal languages, FX($\mathcal{C}$) that capture the essence of the idea of constrained types. By appropriately choosing $\mathcal{C}$, one can get languages that support simple types, just value-dependent types, just type-dependent types, and both. We establish the soundness of the type system for all members of the family.

Outline. The rest of the paper is organized as follows. An informal overview of generic constrained types in X10 is presented in Section 2. Section 3 presents a formal semantics and a proof of soundness. The implementation of generics in X10 is described in Section 4. Section 5 discusses extensions of the type system, including extensions for virtual types and self types. Related work is discussed in Section 6. Finally, Section 7 concludes.

2. X10 language overview

This section presents an informal description of dependent and generic types in X10. As implemented, X10 simplifies these semantics, limiting expressiveness in favor of usability. After using the design presented here to implement part of the X10 runtime library, we found that the semantics are often difficult for the uninitiated to use. These usability problems are explored in Section 2.4. The semantics that follow do not have the restrictions imposed by the X10 compiler; instead, they represent the core of the X10 type system.

X10 is a class-based object-oriented language. The language has a sequential core similar to Java or Scala, but also constructs for concurrency and distribution, as well as constrained types, described here. Like Java, the language provides single class inheritance and multiple interface inheritance.

A constrained type in X10 is written `C{c}`, where `C` is the name of a class and `c` is a constraint on the properties of `C` and the immutable state in scope at the type. The constraint `c` may refer to the value being constrained through the special variable `self`, which has type `C` in the constraint. Constraints are drawn from a constraint language that, syntactically, is a subset of the boolean expressions of X10. For brevity, the constraint may be omitted and interpreted as `true`.

To illustrate the features of dependent types in X10, we develop a `List` class. We will present several versions of `List` as we introduce new features. A `List` class with a type property `T` and an `int` property `len` is declared as in Figure 1. Classes in X10 may be declared with any number of type properties and value properties.

For a class `C` with type properties X_1, …, X_k, we permit the positional syntax $C[T_1,\dots,T_k]$ to mean $C\{X_1==T_1,\dots, X_k==T_k\}$.

Like in Scala, fields are declared using the keywords `var` or `val`. The `List` class has a mutable `head` field with type `T` (which resolves to `this.T`), and an immutable (final) `tail` field with type `List[T]`, that is, with type `List{self.T==this.T}`. Note that `this` occurring in the constraint refers to the instance of the enclosing `List` class,

```
class List[T](len: int) {
  var head: T;
  val tail: List[T];
  def get(i: int): T {
    if (i == 0) return head;
    else return tail.get(i-1);
  }
  def this[S](hd: S, tl: List[S]):
      List[S]{len==tl.len+1} {
    property[S](tl.len+1);
    head = hd; tail = tl;
  }
}
```

Figure 1. List example, simplified

and `self` refers to the value being constrained—`this.tail` in this case.

Methods are declared with the `def` keyword. The method `get` takes a final integer `i` argument and returns the element at that position.

Objects in X10 are initialized with constructors, which 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. X10 uses method syntax with the name `this` for constructors. In X10, constructors have a "return type", which constrains the properties of the new object. The constructor in Figure 1 takes a type argument `S` and two value arguments `hd` and `tl`. The constructor return type specifies that the constructor initializes the object to have type `List[S]{len==tl.len+1}`, that is, `List{self.T==S, self.len==tl.len+1}`. The formal parameter types and return types of both methods and constructors may refer to final parameters of the same declaration.

The body of the constructor begins with a `property` statement that initializes the properties of the new instance. All properties are initialized simultaneously and it is required that the property assignment entail the constructor return type. The remainder of the constructor assigns the fields of the instance with the constructor arguments.

We next illustrate more advanced features of X10 by presenting a new version of `List`, shown in Figure 2.

2.1 Class invariants

Properties of a class may be constrained with a *class invariant*. `List`'s class invariant in Figure 2 specifies that the length of the list is non-negative. The class invariant must be established by all constructors of the class and can subsequently be assumed for all instances of the class.

For generic types, the invariant is used to provide subtyping bounds on the type properties. For example, in the following code, `SortedList` specifies that the element type `T` is a subtype of `Comparable`:

```
class SortedList[T] {T<=Comparable[T]} {
  def sort() { ... x.compare(y) ... }
}
```

```
class List[T](len: int){len >= 0} {
  var head: T;
  val tail: List[T]{self.len==this.len-1};

  def get(i: int{0 <= self, self < len}){len > 0}: T {
    if (i == 0)
      return head;
    val tl = tail as List{len>0};
    val i1 = i-1 as int{0 <= self,self < tail.len};
    return tl.get(i1);
  }

  def map[S](f: T => S): List[S]{self.len==this.len} {
    return new List[S](f(head), tail.map[S](f));
  }

  def this[S](hd: S, tl: List[S]):
      List[S]{len==tl.len+1} {
    property[S](tl.len+1);
    head = hd; tail = tl;
  }

  def print(){T <= Printable} {
    head.print();
    println(", ");
    tail.print();
  }
}
```

Figure 2. List example, with more functionality and more constraints.

2.2 Methods

Method declarations in X10 may have additional constraints, or *guards*, on the method parameters that must be satisfied for access. The guard holds throughout the method body.

The method `get` in Figure 2 has a constraint on the type of `i` that requires that it be within the list bounds. The method also has a guard that requires that the actual receiver's `len` property must be positive—calls to `get` on empty lists are not permitted. A method with a guard is called a *conditional method*. The constraint on `get` ensures that the guard on `tail` is satisfied in the method body. In the method body, the `head` of the list is returned for position `0`; otherwise, the call recurses on `tail`. For this example to type-check, the constraint system must establish the guard on the recursive call to `get`. This is done explicitly by performing run-time casts using the `as` operator.[3]

Method guards can also be constraints on type properties. For instance, the `print` method can be invoked only if `T` is instantiated on a type that implements `Printable`. Since the

[3] Support for flow-sensitive constraints would obviate the need for run-time casts in this context.

guard holds in the body of the method, the print method can be invoked on head—it is guaranteed to implement Printable. This feature is similar to optional methods in CLU [26]. Using structural constraints on T would give more expressive power; optional methods are discussed further in Section 5.

Method overriding is similar to Java: a method of a subclass with the same name and parameter types overrides a method of the superclass. An overridden method may have a return type that is a subtype of the superclass method's return type. A method guard may be weakened by an overriding method; that is, the guard in the superclass must entail the guard in the subclass.

Methods may also have type parameters. For instance, the map method in Figure 2 has a type parameter S and a value parameter that is a function from T to S. A parametrized method is invoked by giving type arguments before the expression arguments (see recursive call to map).[4]

2.3 Type constraints and variance

Type properties and subtyping constraints may be used in X10 to provide *use-site variance* constraints. Use-site variance based on structural virtual types was proposed by Thorup and Torgerson [51] and extended for parametrized type systems by Igarashi and Viroli [22]. The latter type system lead to the development of wildcards in Java [18, 52, 5]. X10's support for use-site variance has similar expressive power. Section 5.2 describes a characterization of wildcards using generic constrained types.

Consider the following subtypes of List from Figure 2.

- List. This type has no constraints on the type property T. Any type that constrains T is a subtype of List. The type List is equivalent to List{true}. For a List xs, the return type of the get method is xs.T. Since the property T is unconstrained, the caller can only assign the return value of get to a variable of type xs.T or of type Object.
- List{T==float}. The type property T is bound to float. For a final expression xs of this type, xs.T and float are equivalent types and can be used interchangeably. The syntax List[float] is used as shorthand for List{T==float}.
- List{T<=Collection}. This type constrains T to be a subtype of Collection. All instances of this type must bind T to a subtype of Collection; for example List[Set] (i.e., List{T==Set}) is a subtype of List{T<=Collection} because T==Set entails T <= Collection. If xs has the type List{T<=Collection}, then the return type of get has type xs.T, which is an unknown but fixed subtype of Collection; the return value can be assigned into a variable of type Collection.
- List{T>=String}. This type bounds the type property T from below. For a List xs of this type, any supertype of String may flow into a variable of type xs.T. The return type of the get method is known to be a supertype of String (and implicitly a subtype of Object).

In the positional syntax for types (e.g., List[T]), an actual type argument T may optionally be annotated with a *use-site variance tag*, either "+" or "-": if X is a type property of C, then the syntax C[+T] is shorthand for C{X<=T} and C[-T] is shorthand for C{X>=T}; of course, C[T] is shorthand for C{X==T}.

2.4 Type properties vs. type parameters

As previously mentioned, after implementing the X10 design above, we encountered some usability problems. The basic problem is that programmers are more familiar with *type parameters* than with *type properties*: providing similar functionality with often subtle distinctions can be difficult for novices users of the language.

The key difference between type parameters and type properties is that type properties are instance members bound during object construction. Type properties are thus accessible through expressions—e.T is a legal type (if e is final)—and are inherited by subclasses. These features gives type properties more expressive power than parameters, but are unfamiliar to many programmers.

One complication is that, since type properties are inherited, the language design needs to account for ambiguities introduced when the same name is used for different type properties declared in or inherited into a class. These can be disambiguated by "casting" the target up to the desired supertype, e.g., (e as C).X specifies the property X inherited from C.

As an example, in the following X10 code, HashMap inherits the properties K and V from Map.

```
interface Map[K,V] {
  def get(K): V;
  def put(K, V): V;
}

class HashMap extends Map {
  def get(k: K): V = ...;
  def put(k: K, v: V): V = ...;
}
```

If a user is used to type parameters, HashMap would be declared as follows:

```
class HashMap[K,V] extends Map[K,V] {
  def get(k: K): V = ...;
  def put(k: K, v: V): V = ...;
}
```

This introduces a new pair of type properties named K and V that shadow the inherited properties, and adds to the class invariant of HashMap the constraint {(this as Map).K==K,

[4] Actual type arguments can be inferred from the types of the value arguments. Type inference is out of the scope of this paper.

(this as Map).V==V}. A naive implementation of type properties would store run-time type information for all four properties in each instance of HashMap.

To correct these problems, a revised X10 design imposes restrictions on type properties so that they behave similarly to type parameters.[5]

We make the syntax e.X illegal and permit type properties to be accessible only from within the body of their class definition via the implicit this qualifier. We require that type constraints be written using the shortened C[T] (or C[+T], etc.) syntax rather than as C{X==T} (C{X<=T}). The compiler also addresses the ambiguous type property problem by ensuring inherited type properties are not visible from subclasses.

With these modifications, explicit type constraints continue to be allowed in method and constructor guards and in class invariants. In this design, unconstrained types (e.g., C) are still legal types.

3. Semantics

We now describe the semantics of languages in the FX family. We start with a core FX(·) language that supports simple FJ-like types, then add value-dependent types and type-dependent types, separately, then finally add both. For uniformity we declare type-valued parameters and properties to be of "type" type, instead of using square brackets to demarcate them.

3.1 FX(·)

The grammar for FX(·) is shown in Figure 3. The syntax is essentially that of FJ. Following the convention of FJ, we use $\bar{x}$ to denote a list $x_1, \ldots, x_n$, and use $\bullet$ to denote the empty list.

A program P is a set of class declarations $\overline{\texttt{L}}$. Class names C range over the declared classes in P and Object. Classes have properties (i.e., immutable fields) $\overline{\texttt{f}}$ and methods $\overline{\texttt{M}}$. We omit constructors and require that the new expression provide initializers for all fields, including inherited fields. Methods are introduced with the def keyword.

Both classes and methods may have constraint clauses c. In the case of classes, c is to be thought of as an *invariant* satisfied by all instances of the class; in the case of methods, c is an additional condition, or *guard*, that must be satisfied by the receiver and the actual arguments of the method in order for the method to be applicable.

Expressions e are either parameters x (including the implicit method parameter this), field accesses, method invocations, new expressions, or casts (written e as T).

The set of types includes classes C and is closed under constrained types (T{c}) and existential quantification ($\exists$x : T. U). A value v is of type C if it is an instance of class C; it is of type T{c} if it is of type T and it satisfies the constraint c[v/self]; it is of type $\exists$x : T. U if there is some value w of type T such that v is of type U[w/x].

The syntax for constraints in FX(·) is specified in Figure 3. As expected, constraints relate property fields of objects. Neither casts nor method invocations are permitted in constraints.

We distinguish a subset of these constraints as *user constraints*—these are permitted to occur in programs. For FX(·) the only user constraint permitted is the vacuous true. Thus the types occurring in user programs are isomorphic to class types, and class and method definitions specialize to the standard class and method definitions of FJ.

The constraints permitted by the syntax in Figure 3 that are not user constraints are used to define the static and dynamic semantics of FX(·) (see, e.g., rule T-FIELD in Figure 8). The use of this richer constraint set as well as constrained and existential types is not necessary in FX(·); it simply enables us to present the static and dynamic semantics once for the entire family of FX languages, specifying the other members of the family as extensions to these core semantics.

Existential constraints are introduced for convenience only: T{$\exists$x : U. c} is equivalent to $\exists$y : U. T{c[y/x]} choosing y not free in T.

Dynamic semantics. The operational semantics, shown in Figure 4, is straightforward and essentially identical to FJ [21]. It is described in terms of a non-deterministic reduction relation on expressions.

The only novelty is the use of the subtyping relation to check that the cast is satisfied. The typing rule for casts (T-NEW) in Figure 8 specifies that if the arguments $\overline{\texttt{e}}$ have type $\overline{\texttt{V}}$ then new C($\overline{\texttt{e}}$) has type $\exists\overline{\texttt{y}} : \overline{\texttt{V}}$. C{self == new C($\overline{\texttt{y}}$)}, therefore R-CAST requires this particular type to be a subtype of T. In FX(·), this test simply involves checking that the class of which the object is an instance is a subclass of the class specified in the given type; in languages with richer notions of type this operation may involve run-time constraint solving using the properties of the object. See Section 5.4 for further discussion of the casts, including decidability issues.

Static Semantics. Each language in the family is defined over a given input constraint system $\mathcal{X}$ that is required to support the trivial constraint true, conjunction, existential quantification, and equality on constraint terms. Given a program P, we now show how to derive from $\mathcal{X}$ a larger deduction system that captures the object-oriented structure of P and lets us decide whether P is well typed.

In the following, the context Γ is always a (finite, possibly empty) sequence of formulas x : T and constraints c satisfying:

1. for any formula ϕ = x : T or constraint ϕ = c in the sequence all free variables y occurring in T or c are declared by a formula y : U in the sequence to the left of ϕ.

[5] We have implemented a further simplification of this new design, moving variance annotations to the definition site.

2. for any variable x, there is at most one formula $\mathtt{x} : \mathtt{T}$ in Γ.

In the judgments that follow, for any formulas ϕ_1 and ϕ_2, we adopt the convention that the rule $\Gamma \vdash \phi_1,\ \phi_2$ is shorthand for the rules $\Gamma \vdash \phi_1$ and $\Gamma \vdash \phi_2$. Whenever we state an assumption of the form "x fresh" in a rule we mean it is not free in the consequent of the rule.

The judgments of interest are as follows:

1. Well-formedness:

$\Gamma \vdash \mathtt{c}\ \mathtt{constraint}$	constraint c is well formed
$\Gamma \vdash \mathtt{T}\ \mathtt{type}$	type T is well formed

2. Member lookup:

$\mathsf{fields}(\mathtt{C}) = \overline{\mathtt{f}} : \overline{\mathtt{T}}$	class C has fields $\overline{\mathtt{f}}$ of type $\overline{\mathtt{T}}$
$\Gamma \vdash \mathtt{C}\ \mathtt{has}\ \mathtt{I}$	class C has member I
$\Gamma \vdash \mathtt{x}\ \underline{\mathtt{has}}\ \mathtt{I}$	variable x has member I

where $\mathtt{I} ::= \mathtt{f} : \mathtt{T} \mid \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{U} = \mathtt{e}$

3. Constraints:

$\Gamma \vdash \mathtt{c}$	constraint c holds

4. Typing:

$\Gamma \vdash \mathtt{e} : \mathtt{T}$	expression e has type T
$\vdash \mathtt{def}\ \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{U} = \mathtt{e}$ OK in C	method m in class C type checks
$\vdash \mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \}$ OK	class C type checks

5. Subtyping:

$\Gamma \vdash \mathtt{S} <: \mathtt{T}$	type S is a subtype of type T

A program type checks iff all its classes do. We now describe in more detail each of these judgments, in turn.

1. Well-formedness. A constraint c is well formed in context Γ, written $\Gamma \vdash \mathtt{c}\ \mathtt{constraint}$, iff both its free variables are declared in Γ and it is well formed for the underlying constraint system. The rules in Figure 5 extend this requirement to types.

We say a program, context, or judgment is well formed iff all the constraints and types involved are well formed. By design, every judgment derived from a well-formed context is also well formed. As a consequence, if a program type checks, it is well formed.

2. Member lookup. Figure 6 specifies the fields and methods available on each class. We impose restrictions on inheritance as follows. Classes may only be extended by classes with stronger invariants. Fields cannot be overridden. Methods cannot be overloaded. A method may only be overridden by a method with the same name, arity, parameter types, a return type that is a subtype of the overridden return type, and a weaker guard (i.e., the superclass method guard must entail the subclass's). [6]

To prepare for the introduction of generic types later, we distinguish members that are available on variables from members available on classes. For now, $\mathtt{x} : \mathtt{C}\ \underline{\mathtt{has}}\ \mathtt{I}[\mathtt{x}/\mathtt{this}]$ iff $\mathtt{C}\ \mathtt{has}\ \mathtt{I}$.

3. Constraints. In defining these judgments we will use $\Gamma \vdash_{\mathcal{X}} \mathtt{c}$, the judgment corresponding to the underlying constraint system. Formally, $\mathcal{X}$ permits judgments of the form $\Gamma \vdash_{\mathcal{X}} \mathtt{c}$ where $\Gamma ::= \overline{\mathtt{c}}$ is a (finite, possibly empty) sequence of constraints. We define the *constraint projection*, $\sigma(\Gamma)$ as follows.

$$\begin{array}{l}
\sigma(\varepsilon) = \mathtt{true} \\
\sigma(\mathtt{x} : \mathtt{C}, \Gamma) = \sigma(\Gamma) \\
\sigma(\mathtt{x} : \mathtt{T}\{\mathtt{c}\}, \Gamma) = \mathtt{c}[\mathtt{x}/\mathtt{self}], \sigma(\mathtt{x} : \mathtt{T}, \Gamma) \\
\sigma(\mathtt{x} : \exists \mathtt{y} : \mathtt{T}.\ \mathtt{U}, \Gamma) = \sigma(\mathtt{z} : \mathtt{T}, \mathtt{x} : \mathtt{U}[\mathtt{z}/\mathtt{y}], \Gamma) \\
\sigma(\mathtt{c}, \Gamma) = \mathtt{c}, \sigma(\Gamma)
\end{array}$$

Above, in the fourth rule, we assume that alpha-equivalence is used to choose a variable z that does not occur in the context under construction.

We define $\Gamma \vdash \mathtt{c}$ in Figure 7. X-SEL permits the underlying constraint system $\mathcal{X}$ to interpret field accesses. X-INV handles class invariants. Note the use of subtyping here to prepare for the introduction of bounds on generics types later.

We say that a context Γ is *consistent* if all (finite) subsets of $\{\mathtt{c} \mid \sigma(\Gamma) \vdash \mathtt{c}\}$ are consistent. In all type judgments presented below (T-CAST, T-FIELD, etc.), we make the implicit assumption that the context Γ is consistent; if it is inconsistent, the rule cannot be used and the type of the given expression cannot be established (type-checking fails).

4. Typing. The typing rules are specified in Figure 8.

T-VAR is as expected, except that it asserts the constraint self==x, which records that any value of this type is known statically to be equal to x. This constraint is actually very crucial—as we shall see in the other rules, once we establish that an expression e is of a given type T, we "transfer" the type to a freshly chosen variable z. If, in fact, e has a static "name" x (i.e., e is known statically to be equal to x; that is, it has type T{self==x}), then T-VAR lets us assert that z:T{self==x}, i.e., that z equals x. Thus T-VAR provides an important base case for reasoning statically about equality of values in the environment.

We do away with the three cast rules in FJ in favor of a single cast rule, requiring only that e be of some type U. At run time, e will be checked to see if it is actually of type T (see R-CAST in Figure 4).

T-FIELD may be understood through "proxy" reasoning as follows: Given the context Γ, assume the receiver e can be established to be of type T. Now, we do not know the runtime value of e, so we shall assume that it is some fixed but unknown "proxy" value x (of type T) that is "fresh" in that it is not known to be related to any known value (i.e., those recorded in Γ). If we can establish that x has a field f of type U[7], then we can assert that e.f has type U and, further,

[6] To avoid cluttering the inference rules, we define overriding only informally; a formal definition is straightforward.

[7] Note from the definition of fields in Figure 6 that all occurrences of this in the declared type of the field f will have been replaced by x.

$$\begin{array}{rlll}
\text{(Program)} & \mathtt{P} & ::= & \overline{\mathtt{L}} \\
\text{(Class declaration)} & \mathtt{L} & ::= & \mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \\
\text{(Method declaration)} & \mathtt{M} & ::= & \mathtt{def}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}:\mathtt{T} = \mathtt{e}; \\
\text{(Expression)} & \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{e}\ \mathtt{as}\ \mathtt{T} \\
\text{(Value)} & \mathtt{v} & ::= & \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}) \\
\text{(Type)} & \mathtt{T} & ::= & \mathtt{C} \mid \mathtt{T}\{\mathtt{c}\} \mid \exists \mathtt{x}:\mathtt{T}.\,\mathtt{T} \\
\text{(Constraint term)} & \mathtt{t} & ::= & \mathtt{x} \mid \mathtt{t.f} \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{t}}) \\
\text{(Constraint)} & \mathtt{c} & ::= & \mathtt{true} \mid \mathtt{t} == \mathtt{t} \mid \mathtt{c},\mathtt{c} \mid \exists \mathtt{x}:\mathtt{T}.\,\mathtt{c}
\end{array}$$

Figure 3. FX productions. C ranges over class names, f over field names, m over method names, x over variable names.

$$\frac{\mathsf{fields}(\mathtt{C}) = \overline{\mathtt{f}}:\overline{\mathtt{T}}}{\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{f}_i \rightarrow \mathtt{t}_i}\ \text{(R-FIELD)} \qquad \frac{\mathtt{e}_i \rightarrow \mathtt{e}'_i}{\mathtt{new}\ \mathtt{C}(\ldots,\mathtt{e}_i,\ldots) \rightarrow \mathtt{new}\ \mathtt{C}(\ldots,\mathtt{e}'_i,\ldots)}\ \text{(RC-NEW-ARG)}$$

$$\frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{e.f} \rightarrow \mathtt{e}'.\mathtt{f}}\ \text{(RC-FIELD)} \qquad \frac{\mathtt{C}\ \mathsf{has}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}:\mathtt{U} = \mathtt{b}}{\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{a}}) \rightarrow \mathtt{b}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}),\overline{\mathtt{a}}/\mathtt{this},\overline{\mathtt{x}}]}\ \text{(R-INVK)}$$

$$\frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{e.m}(\overline{\mathtt{a}}) \rightarrow \mathtt{e}'.\mathtt{m}(\overline{\mathtt{a}})}\ \text{(RC-INVK-RECV)} \qquad \frac{\mathtt{a}_i \rightarrow \mathtt{a}'_i}{\mathtt{e.m}(\ldots,\mathtt{a}_i,\ldots) \rightarrow \mathtt{e.m}(\ldots,\mathtt{a}'_i,\ldots)}\ \text{(RC-INVK-ARG)}$$

$$\frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{e}\ \mathtt{as}\ \mathtt{T} \rightarrow \mathtt{e}'\ \mathtt{as}\ \mathtt{T}}\ \text{(RC-CAST)} \qquad \frac{\vdash \overline{\mathtt{e}}:\overline{\mathtt{V}} \qquad \vdash \exists \overline{\mathtt{y}}:\overline{\mathtt{V}}.\,\mathtt{C}\{\mathtt{self} == \mathtt{new}\ \mathtt{C}(\overline{\mathtt{y}})\} <: \mathtt{T}}{\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})\ \mathtt{as}\ \mathtt{T} \rightarrow \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})}\ \text{(R-CAST)}$$

Figure 4. FX operational semantics

$$\vdash \mathtt{Object}\ \mathtt{type}\ \text{(W-OBJECT)} \qquad \frac{\Gamma \vdash \mathtt{T}\ \mathtt{type} \qquad \Gamma, \mathtt{self}:\mathtt{T} \vdash \mathtt{c}\ \mathtt{constraint}}{\Gamma \vdash \mathtt{T}\{\mathtt{c}\}\ \mathtt{type}}\ \text{(W-DEP)}$$

$$\frac{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P}}{\vdash \mathtt{C}\ \mathtt{type}}\ \text{(W-CLASS)} \qquad \frac{\Gamma \vdash \mathtt{T}\ \mathtt{type} \qquad \Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{U}\ \mathtt{type}}{\Gamma \vdash \exists \mathtt{x}:\mathtt{T}.\,\mathtt{U}\ \mathtt{type}}\ \text{(W-EXISTS)}$$

Figure 5. FX well-formed types

$$\vdash \mathsf{fields}(\mathtt{Obj}) = \bullet\ \text{(L-FIELDS-OBJ)} \qquad \frac{\vdash \mathsf{fields}(\mathtt{C}) = \overline{\mathtt{f}}:\overline{\mathtt{T}}}{\mathtt{C}\ \mathsf{has}\ \mathtt{f}_i:\mathtt{T}_i}\ \text{(L-FIELD-B)} \qquad \frac{\mathtt{C}\ \mathsf{has}\ \mathtt{I}}{\mathtt{x}:\mathtt{C} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{I}[\mathtt{x}/\mathtt{this}]}\ \text{(L-MEMBER-B)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}, \mathtt{c} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{I}}{\Gamma, \mathtt{x}:\mathtt{T}\{\mathtt{c}\} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{I}}\ \text{(L-MEMBER-C)} \qquad \frac{\Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{I} \qquad \mathtt{y}\ \text{fresh}}{\Gamma, \mathtt{x}:\exists \mathtt{y}:\mathtt{U}.\,\mathtt{T} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{I}}\ \text{(L-MEMBER-E)}$$

$$\frac{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \qquad \vdash \mathsf{fields}(\mathtt{D}) = \overline{\mathtt{g}}:\overline{\mathtt{U}}}{\vdash \mathsf{fields}(\mathtt{C}) = \overline{\mathtt{g}}:\overline{\mathtt{U}}, \overline{\mathtt{f}}:\overline{\mathtt{T}}}\ \text{(L-FIELDS-I)}$$

$$\frac{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \qquad \mathtt{def}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{U}})\{\mathtt{d}\}:\mathtt{V} = \mathtt{e} \in \overline{\mathtt{M}}}{\vdash \mathtt{C}\ \mathsf{has}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{U}})\{\mathtt{d}\}:\mathtt{V} = \mathtt{e}}\ \text{(L-METHOD-B)}$$

$$\frac{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \qquad \vdash \mathtt{D}\ \mathsf{has}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{U}})\{\mathtt{d}\}:\mathtt{V} = \mathtt{e} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\vdash \mathtt{C}\ \mathsf{has}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{U}})\{\mathtt{d}\}:\mathtt{V} = \mathtt{e}}\ \text{(L-METHOD-I)}$$

Figure 6. FX member lookup

$$\frac{\sigma(\Gamma) \vdash_{\mathcal{X}} \mathtt{c}}{\Gamma \vdash \mathtt{c}} \quad \text{(X-PROJ)} \qquad \frac{\vdash \mathsf{fields}(\mathtt{C}) = \overline{\mathtt{f}} : \overline{\mathtt{T}} \quad \Gamma \vdash \mathsf{new}\ \mathtt{C}(\overline{\mathtt{t}}) : \mathtt{U} \quad \Gamma, \mathsf{new}\ \mathtt{C}(\overline{\mathtt{t}}).\overline{\mathtt{f}} == \overline{\mathtt{t}} \vdash \mathtt{c}}{\Gamma \vdash \mathtt{c}} \quad \text{(X-SEL)}$$

$$\frac{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \quad \Gamma \vdash \mathtt{t} : \mathtt{U},\ \mathtt{U} <: \mathtt{C} \quad \Gamma, \mathtt{c}[\mathtt{t}/\mathtt{this}] \vdash \mathtt{d}}{\Gamma \vdash \mathtt{d}} \quad \text{(X-INV)}$$

Figure 7. FX object constraint system

$$\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}\{\mathtt{self} == \mathtt{x}\} \quad \text{(T-VAR)} \qquad \frac{\Gamma \vdash \mathsf{e} : \mathtt{U} \quad \Gamma \vdash \mathtt{T}\ \mathsf{type}}{\Gamma \vdash \mathsf{e}\ \mathsf{as}\ \mathtt{T} : \mathtt{T}} \quad \text{(T-CAST)}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathtt{T} \quad \Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{f} : \mathtt{U} \quad \mathtt{x}\ \text{fresh}}{\Gamma \vdash \mathsf{e.f} : \exists \mathtt{x} : \mathtt{T}.\ \mathtt{U}\{\mathtt{self} == \mathtt{x.f}\}} \quad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathtt{T},\ \overline{\mathtt{a}} : \overline{\mathtt{U}} \quad \Gamma, \mathtt{x} : \mathtt{T}, \overline{\mathtt{y}} : \overline{\mathtt{U}} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{m}(\overline{\mathtt{y}} : \overline{\mathtt{V}})\{\mathtt{d}\} : \mathtt{W} = \mathtt{b},\ \overline{\mathtt{U}} <: \overline{\mathtt{V}},\ \mathtt{d} \quad \mathtt{x}, \overline{\mathtt{y}}\ \text{fresh}}{\Gamma \vdash \mathsf{e.m}(\overline{\mathtt{a}}) : \exists \mathtt{x} : \mathtt{T}.\ \exists \overline{\mathtt{y}} : \overline{\mathtt{U}}.\ \mathtt{W}} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \quad \mathsf{fields}(\mathtt{C}) = \overline{\mathtt{g}} : \overline{\mathtt{U}} \\ \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \quad \Gamma, \mathtt{x} : \mathtt{C}, \overline{\mathtt{y}} : \overline{\mathtt{V}}, \mathtt{x}.\overline{\mathtt{g}} == \overline{\mathtt{y}} \vdash \overline{\mathtt{V}} <: \overline{\mathtt{U}}[\mathtt{x}/\mathtt{this}],\ \mathtt{c}[\mathtt{x}/\mathtt{this}] \quad \mathtt{x}, \overline{\mathtt{y}}\ \text{fresh}\end{array}}{\Gamma \vdash \mathsf{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \exists \overline{\mathtt{y}} : \overline{\mathtt{V}}.\ \mathtt{C}\{\mathtt{self} == \mathsf{new}\ \mathtt{C}(\overline{\mathtt{y}})\}} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \quad \mathsf{def}\ \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{U}})\{\mathtt{d}\} : \mathtt{V} = \mathsf{e} \in \overline{\mathtt{M}} \\ \mathtt{this} : \mathtt{C}, \overline{\mathtt{x}} : \overline{\mathtt{U}} \vdash \overline{\mathtt{U}}\ \mathsf{type},\ \mathtt{d}\ \mathsf{constraint} \quad \mathtt{this} : \mathtt{C}, \overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{d} \vdash \mathsf{e} : \mathtt{W},\ \mathtt{W} <: \mathtt{V}\end{array}}{\vdash \mathsf{def}\ \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{U}})\{\mathtt{d}\} : \mathtt{V} = \mathsf{e}\ \text{OK in}\ \mathtt{C}} \quad \text{(OK-METHOD)}$$

$$\frac{\mathtt{this} : \mathtt{C}, \overline{\mathtt{f}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{T}}\ \mathsf{type},\ \mathtt{c}\ \mathsf{constraint} \quad \vdash \mathtt{D}\ \mathsf{type} \quad \overline{\mathtt{M}}\ \text{OK in}\ \mathtt{C}}{\vdash \mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \}\ \text{OK}} \quad \text{(OK-CLASS)}$$

Figure 8. FX typing rules

$$\frac{\Gamma \vdash \mathtt{T}\ \mathsf{type}}{\Gamma \vdash \mathtt{T} <: \mathtt{T}} \quad \text{(S-REFL)} \qquad \frac{\Gamma \vdash \mathtt{T} <: \mathtt{U}, \mathtt{U} <: \mathtt{V}}{\Gamma \vdash \mathtt{T} <: \mathtt{V}} \quad \text{(S-TRANS)} \qquad \frac{\Gamma \vdash \mathtt{T}\{\mathtt{c}\}\ \mathsf{type} \quad \Gamma, \mathtt{c} \vdash \mathtt{T} <: \mathtt{U}}{\Gamma \vdash \mathtt{T}\{\mathtt{c}\} <: \mathtt{U}} \quad \text{(S-CONST-L)}$$

$$\frac{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P}}{\Gamma \vdash \mathtt{C} <: \mathtt{D}} \quad \text{(S-CLASS)} \qquad \frac{\Gamma \vdash \mathtt{U}\{\mathtt{c}\}\ \mathsf{type} \quad \Gamma, \mathtt{self} : \mathtt{T} \vdash \mathtt{c}, \mathtt{T} <: \mathtt{U}}{\Gamma \vdash \mathtt{T} <: \mathtt{U}\{\mathtt{c}\}} \quad \text{(S-CONST-R)}$$

$$\frac{\Gamma \vdash \mathtt{T}\ \mathsf{type} \quad \Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{U} <: \mathtt{V} \quad \mathtt{x}\ \text{fresh}}{\Gamma \vdash \exists \mathtt{x} : \mathtt{T}.\ \mathtt{U} <: \mathtt{V}} \quad \text{(S-EXISTS-L)} \qquad \frac{\Gamma \vdash \mathtt{t} : \mathtt{T},\ \mathtt{U} <: \mathtt{V}[\mathtt{t}/\mathtt{x}]}{\Gamma \vdash \mathtt{U} <: \exists \mathtt{x} : \mathtt{T}.\ \mathtt{V}} \quad \text{(S-EXISTS-R)}$$

Figure 9. FX subtyping rules

$$\vdash \mathsf{type}\ \mathsf{type} \quad \text{(W-TYPE)} \qquad \Gamma, \mathtt{x} : \mathsf{type} \vdash \mathtt{x}\ \mathsf{type} \quad \text{(W-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{T} <= \mathtt{U} \quad \Gamma \vdash \mathtt{T} : \mathsf{type},\ \mathtt{U} : \mathsf{type}}{\Gamma \vdash \mathtt{T} <: \mathtt{U}} \quad \text{(S-EXTENDS)} \qquad \frac{\Gamma \vdash \mathsf{p} : \mathtt{T} \quad \Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{f} : \mathsf{type}}{\Gamma \vdash \mathsf{p.f}\ \mathsf{type}} \quad \text{(W-PATH)}$$

$$\frac{\Gamma \vdash \mathtt{T} <= \mathtt{U} \quad \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{I}}{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x}\ \underline{\mathsf{has}}\ \mathtt{I}} \quad \text{(L-EXTENDS)} \qquad \frac{\Gamma \vdash \mathtt{C}\{\mathtt{c}\}\ \mathsf{type}}{\Gamma \vdash \mathtt{C}\{\mathtt{c}\} : \mathsf{type}\{\mathtt{self} == \mathtt{C}\{\mathtt{c}\}\}} \quad \text{(T-TYPE)}$$

Figure 10. $\mathsf{FX}(\mathcal{G})$

that it equals `x.f`. Hence, we can assert that `e.f` has type $\exists \mathtt{x}:\mathtt{T}.\,\mathtt{U}\{\mathtt{self}=\mathtt{x.f}\}$.

T-INVK has a similar structure to T-FIELD: we use proxy reasoning for the receiver and the arguments of the method call. T-NEW also uses the same proxy reasoning: however in this case we can establish that the resulting value is equal to $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{y}})$ for some values $\overline{\mathtt{y}}$ of the given types.

OK-METHOD and OK-CLASS ensure that the types and constraints occurring in a program are well formed. Following from FJ, these rules do not preclude the existence of cycles in the type declarations. We assume they are acyclic. OK-METHOD checks that the actual type `W` of the method body `e` is a subtype of its declared return type `V`. OK-CLASS makes sure all methods of the class type check.

5. Subtyping. The subtyping relation is defined in Figure 9. Unsurprisingly, it is reflexive (S-REFL) and transitive (S-TRANS). S-EXISTS-L and S-EXISTS-R handle existential types. S-CONST-L and S-CONST-R handle constraints. The rules ensure all types are well formed.

3.2 FX($\mathcal{A}$)

Turning FX($\cdot$) into a language with value-dependent types is straightforward since the construction of the previous section is parametric in the underlying constraint system $\mathcal{X}$, and constraint propagation is already built into the typing rules.

First, we assume we are given a constraint system $\mathcal{A}$ with a vocabulary of primitive types `A`, predicates `p`, functions `q`, and literals `l`.

Second, we extend the productions of FX($\cdot$) as follows.

$$\begin{array}{rlll}
\text{(Type)} & \mathtt{T} & ::= & \mathtt{A} \\
\text{(Expression)} & \mathtt{e} & ::= & \mathtt{q}(\overline{\mathtt{e}}) \\
\text{(Values)} & \mathtt{v} & ::= & \mathtt{l} \\
\text{(Constraint term)} & \mathtt{t} & ::= & \mathtt{q}(\overline{\mathtt{t}}) \\
\text{(Constraint)} & \mathtt{c} & ::= & \mathtt{p}(\overline{\mathtt{t}})
\end{array}$$

The following rules are needed to type functions and literals:

$$\frac{\mathtt{q}\text{ has type }\overline{\mathtt{A}}\to\mathtt{B}\text{ in }\mathcal{A} \qquad \Gamma\vdash\overline{\mathtt{e}}:\overline{\mathtt{A}}}{\Gamma\vdash\mathtt{q}(\overline{\mathtt{e}}):\exists\overline{\mathtt{x}}:\overline{\mathtt{A}}.\,\mathtt{B}\{\mathtt{self}==\mathtt{q}(\overline{\mathtt{x}})\}} \quad \text{(T-FUN)}$$

$$\frac{\mathtt{l}\text{ has type }\mathtt{A}\text{ in }\mathcal{A}}{\Gamma\vdash\mathtt{l}:\mathtt{A}\{\mathtt{self}==\mathtt{l}\}} \quad \text{(T-LIT)}$$

We extend the operational semantics with rules:

$$\frac{\mathtt{e}_i\to\mathtt{e}_i'}{\mathtt{q}(\ldots,\mathtt{e}_i,\ldots)\to\mathtt{q}(\ldots,\mathtt{e}_i',\ldots)} \quad \text{(RC-FUN)}$$

$$\frac{\mathtt{q}\text{ has type }\overline{\mathtt{A}}\to\mathtt{B}\text{ in }\mathcal{A} \qquad \Gamma\vdash\overline{\mathtt{v}}:\overline{\mathtt{A}}}{\mathtt{q}(\overline{\mathtt{v}})\to[\![\mathtt{q}(\overline{\mathtt{v}})]\!]} \quad \text{(R-FUN)}$$

where $[\![\mathtt{q}(\overline{\mathtt{v}})]\!]$ denotes the result of the evaluation of function `q` on values $\overline{\mathtt{v}}$.

Finally, we permit users to write constraints in $\mathcal{A}$ (except for existential constraints) in programs. For instance, if $\mathcal{A}$ defines the type `int`, integer literals, and the usual arithmetic operators, then we can declare:

```
class Count(n:int) extends Object {
  def inc():Count{self.n==this.n+1} =
    new Count(this.n+1);
}
```

In practice, it makes sense to distinguish the functions of the constraint language from the functions of the base language. One would define the T-FUN typing judgment on a case-by-case basis to relate the interpretation of `q` as an expression to its interpretation as a constraint term.

FX($\mathcal{A}$) corresponds to the CFJ calculus presented in our prior work on constrained types [40]. As described there, X10 supports equality constraints and has been extended with constraint systems for Presburger arithmetic and for set constraints over X10's array index domains (viz., regions).

3.3 FX($\mathcal{G}$)

We now turn to showing how FGJ-style generics can be supported in the FX family. FX($\mathcal{G}$) is the language obtained by adding to FX($\cdot$) the following productions:

$$\begin{array}{rlll}
\text{(Expression)} & \mathtt{e} & ::= & \mathtt{C}\{\mathtt{c}\} \\
\text{(Value)} & \mathtt{v} & ::= & \mathtt{C}\{\mathtt{c}\} \\
\text{(Path)} & \mathtt{p} & ::= & \mathtt{x} \mid \mathtt{p.f} \\
\text{(Type)} & \mathtt{T} & ::= & \mathtt{p} \mid \mathtt{type} \\
\text{(Constraint term)} & \mathtt{t} & ::= & \mathtt{T} \\
\text{(Constraint)} & \mathtt{c} & ::= & \mathtt{T} <= \mathtt{T}
\end{array}$$

and deduction rules of Figure 10.

First we introduce the "type" `type`. FGJ method type parameters are modeled in FX($\mathcal{G}$) as normal parameters of type `type`.[8] Generic class parameters are modeled as ordinary fields of type `type`, with parameter bound information recorded as a constraint in the class invariant. This decision to use fields rather than parameters is discussed further in Section 2.4. In brief, it permits powerful idioms using fixed but unknown types without requiring "wildcards".

The set of well-formed types is now enhanced to permit some fixed but unknown types `x` as well as *path types* (cf. [41]), i.e., type-valued fields of objects as types.[9] We extend σ in the obvious way:

$$\sigma(\mathtt{x}:\mathtt{type},\Gamma)=\sigma(\Gamma)$$
$$\sigma(\mathtt{x}:\mathtt{y},\Gamma)=\sigma(\Gamma)$$
$$\sigma(\mathtt{x}:\mathtt{p.f},\Gamma)=\sigma(\Gamma)$$

[8] In concrete X10 syntax type parameters are distinguished from ordinary value parameters through the use of "square" brackets. This is particularly useful in implementing type inference for generic parameters. We abstract these concerns away in the abstract syntax presented in this section.

[9] But we will not permit invocations of methods with return type `type` to be used as types. This does indeed make sense, but developing this theory further is beyond the scope of this paper.

Reciprocally, we permit class types $\mathtt{C\{c\}}$ to be used as expressions. We type them accordingly (T-TYPE). In contrast, the "type" `type` is neither a valid expression nor a class type: it has no field, method, subclass, or superclass. It may however be constrained as usual as, for instance, in rule T-TYPE; that is to say, we permit equality constraints over types.[10]

The key idea is that information about type-valued expressions can be accumulated through constraints. Specifically we introduce the "extends" constraint `T <= U`. It may be used, for instance, to specify upper bounds on type variables or fields (path types). In $\mathsf{FX}(\mathcal{G})$, users are permitted to specify "==" and "<=" constraints about type variables, fields, and class types.

EXAMPLE 3.1. *The* FGJ *parametric method*

```
<T> T id(T x) { return x; }
```

can be represented as

```
def id(T: type, x: T): T = x;
```

EXAMPLE 3.2. *The* FGJ *class*

```
class Comparator<B> {
  int compare(B y) { ... } }
class SortedList<T extends Comparator<T>> {
  int m(T x, T y) { return x.compare(y); } }
```

can be represented as

```
class Comparator(B: type) {
  def compare(y:B):int = ...; }
class SortedList(T: type)
    {T <= Comparator{self.B==T}} {
  def m(x:T, y:T):int = x.compare(y); }
```

We require the underlying constraint system $\mathcal{G}$ to treat "<=" as a partial order relation (reflexive, antisymmetric, and transitive). It is possible for a program to specify constraints incompatible with the class hierarchy, e.g., `x <= C` and `x <= D` if both class `C` and class `D` extend `Object`. We therefore require $\mathcal{G}$ to treat as inconsistent all sets of constraints on type-valued variables that admit no valuations where these variables take on types as values.

The "<=" constraint is used in two deduction rules. If type `T` extends type `U`, then

- S-EXTENDS. `T` is a subtype of `U`. A method or constructor with argument type `U` may be passed a parameter of type `T`.
- L-EXTENDS. If `x` has type `U` then `x` has all the members of type `T`. Note we only extend the "$\underline{\mathtt{has}}$" predicate that is used in typing judgments. On the other hand, the "`has`" predicate used for method lookup in the operational semantics is not affected.

[10] Type equality is just equality over uninterpreted functions.

The modification of the lookup predicate is necessary to permit typing method invocations with receivers of generic types. It has the unfortunate side effect that we can no longer ensure that type derivations—and hence types—are unique. For instance, given the class definitions:

```
class A() extends Object { def m():A = new A(); }
class B() extends A { def m():B new B(); }
class C(f:type){this.f<=A} extends Object {}
class D(){this.f<=B} extends C { ..this.f.m().. }
```

occurrences of `this.f` in `D` are bounded both by `A` and `B` hence `this.f.m()` may either be typed using the declaration of `m` in `A` or `B`.

Another property of $\mathsf{FX}(\mathcal{G})$ worth noticing is that casts can "erase" typing information. Consider the program:

```
class C() extends Object {}
class D(f:type, g:this.f) extends Object {}
```

Class `D` has a type parameter `f` and a value field `g` of type `f`. Thanks to constraints, if $\mathtt{e} = \mathtt{new\ D(C,new\ C())}$, then expression `e.g` can be shown to have type `C`. In contrast $(\mathtt{e\ as\ D}).\mathtt{g}$ has type $\exists \mathtt{x} : \mathtt{D}.\mathtt{x}.\mathtt{f}\{\mathtt{self} == \mathtt{x}.\mathtt{g}\}$. The type of $(\mathtt{e\ as\ D}).\mathtt{g}$ is essentially "unknown" because the cast erased all information about it. In X10, we choose to shield users from existential types and only permit casts of the form $(\mathtt{e\ as\ D}\{\mathtt{self.f} == \mathtt{t}\})$ where `t` is a type in scope (class type, type parameter, or path type).

3.4 $\mathsf{FX}(\mathcal{G},\mathcal{A})$

No additional rules are needed beyond those of $\mathsf{FX}(\mathcal{G})$ and $\mathsf{FX}(\mathcal{A})$. This language permits type and value constraints, supporting FGJ style generics and value-dependent types. All constraints but existential constraints are now user constraints.

3.5 Results

The following results hold for $\mathsf{FX}(\mathcal{G},\mathcal{A})$ supposing the program `P` type checks.

THEOREM 3.1 (Subject Reduction). *If* Γ *is well formed and* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$, *then for some type* `S`, $\Gamma \vdash \mathtt{e}' : \mathtt{S}$, $\mathtt{S} <: \mathtt{T}$.

Values are of the form $\mathtt{v} ::= \mathtt{new\ C}(\overline{\mathtt{v}}) \mid \mathtt{l} \mid \mathtt{C\{c\}}$.

THEOREM 3.2 (Progress). *If* $\vdash \mathtt{e} : \mathtt{T}$ *then one of the following conditions holds:*

1. `e` *is a value,*
2. `e` *contains a cast sub-expression that is stuck,*
3. *there exists an* $\mathtt{e}'$ *s.t.* $\mathtt{e} \rightarrow \mathtt{e}'$.

THEOREM 3.3 (Type soundness). *If* $\vdash \mathtt{e} : \mathtt{T}$ *and* `e` *reduces to a normal form* $\mathtt{e}'$, *then either* $\mathtt{e}'$ *is a value* `v` *and* $\vdash \mathtt{v} : \mathtt{S}, \mathtt{S} <: \mathtt{T}$ *or* $\mathtt{e}'$ *contains a stuck cast sub-expression.*

Proof sketch. The proof of the same results for a formal language essentially equivalent to $\mathsf{FX}(\mathcal{A})$ has been reported in [40]. We discuss here the key insights that permit us to revise this proof in order to encompass $\mathsf{FX}(\mathcal{G},\mathcal{A})$.

- Subject reduction. Having potentially multiple types for an expression does not make the proof any harder as the subject reduction theorem lets us choose $\mathtt{S}$ among the possible types of $\mathtt{e}'$.

  The main novelty of the $\mathsf{FX}(\mathcal{G})$ type system is that it permits the $\underline{\mathtt{has}}$ predicate to look for methods in arbitrary superclasses or upper bounds of the type under scrutiny. This is not so much a concern for fields as they cannot be overridden. Because methods can, we must adapt the proof of subject reduction for the execution step corresponding to a method invocation (R-INVK).

  First, we observe that the operational semantics rule for method invocations (R-INVK) is required to employ the "correct" method for objects of class C, that is, the first method m found on the inheritance path from class C to class Object from the bottom up. Second, thanks to overriding restrictions, we know that this method must have a return type that is a subtype of any other method m defined in any superclass of C. Finally, because constraint sets incompatible with the class hierarchy are inconsistent, we also know that the type of $\mathtt{new\ C}(\bar{\mathtt{e}})$ cannot be constrained to have any upper bound that is not C itself or one of its superclasses.

  We therefore derive that any method instance one could use to type the expression $\mathtt{new\ C}(\bar{\mathtt{e}}).\mathtt{m}(\bar{\mathtt{a}})$ has a return type that is a supertype of the return type of the only method instance that can be used to make a step of execution. We assume the program type checks; hence, by OK-METHOD, we know that the actual residue $\mathtt{b}[\mathtt{new\ C}(\bar{\mathtt{e}}),\bar{\mathtt{a}}/\mathtt{this},\bar{\mathtt{x}}]$ is guaranteed to have a type that is a subtype of its declared type. Therefore, by transitivity of the subtyping relation, we can derive that if $\mathtt{T}$ is a type of $\mathtt{new\ C}(\bar{\mathtt{e}}).\mathtt{m}(\bar{\mathtt{a}})$, then there exists a type $\mathtt{S}$ of $\mathtt{b}[\mathtt{new\ C}(\bar{\mathtt{e}}),\bar{\mathtt{a}}/\mathtt{this},\bar{\mathtt{x}}]$ that is a subtype of $\mathtt{T}$.

- Progress. $\mathsf{FX}(\mathcal{G},\mathcal{A})$ only differs from $\mathsf{FX}(\mathcal{A})$ in that it admits a new kind of expressions: $\mathtt{C}\{\mathtt{c}\}$. But these are also values, so the proof of progress is essentially unchanged.
- Type soundness. Direct consequence of the previous two theorems.

4. Translation

This section describes an implementation approach for generic types in X10 on a JVM, with bytecode rewriting.

The design is a hybrid design combining techniques of run-time instantiation from NextGen [6, 1] and type-passing from PolyJ [35]. Generic classes are translated into "template" classes that are instantiated on demand at run time by binding the type properties to concrete types. Constraints on values are erased from type references. Adapter objects are used to represent type properties and constraints. Run-time type tests (e.g., casts) are translated into code that checks those constraints at run time. This design has been implemented in the X10 compiler, built on the Polyglot compiler framework [38]. The compiler translates X10 source to Java source, which is then compiled to Java bytecode using an off-the-shelf Java compiler.[11] The X10 runtime is augmented with a class loader implementation that performs run-time instantiation.

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 its bytecode, specializing it to the constraint c. The implementation specializes code based on type constraints, not value constraints; we leave value-constraint specialization to future work. For example, consider the following classes.

```
class A[X] {
  var a: X;
}

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

The compiler generates the following Java code:

```
@Parameters({"X"})
class A {
  @TypeProperty public static class X { }
  public x10.runtime.Type X;
  X a;
  @Synthetic public A(Class X) { this(); }
}

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

The member class A.X is used in place of the type property X. The field X of type x10.runtime.Type captures the actual constrained type on which A is instantiated, and is used for run-time type tests. The @Parameters annotation on A is used during run-time instantiation to identify the type properties. Synthetic constructors with added Class parameters are used to pass instantiation arguments to the new expression. This code is compiled to Java bytecode.

When an expression (e.g., new C()) is evaluated, the class C is 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;
}
```

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

The class loader rewrites allocations of template classes (e.g., `new A(int.class)`) into allocations of the instantiated classes (i.e., `new A$$int()`). The template class name and actual type arguments are mangled to derive the name of the instantiated class. This code cannot be generated directly because class `A$$int` does not yet exist; the Java source compiler would fail to compile `C`.

Upon evaluation of the constructor, the class `A$$int` is 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 `X` with the concrete type `int`.

Parameter types are coerced to and from the actual type `T` (a Java primitive type or `Object`) using method `Runtime.to$T(Object)` and `Runtime.from(T)`, possibly with additional casts. Both are eliminated from the transformed bytecode, but are needed for the template class to type-check.

Passing type arguments. For types visible at run time, annotations are used to pass actual type arguments to the class loader. The annotation `@InstantiateClass` specifies the type parameter and is placed on fields, methods, method parameters, and classes to indicate instantiation parameters for field types, method return types, method parameters, and superclasses, respectively. Interface instantiations are similarly handled by `@InstantiateInterfaces`. The annotation `@Instantiation` is used for parametrized exceptions. The class loader uses the arguments of the annotations to propagate the instantiation information of the enclosing class to the instantiation of annotated entities. It then turns these entities into references to the appropriate dynamically instantiated classes.

Type arguments are passed to allocation expressions as synthetic constructor arguments. Run-time type tests and casts receive type parameters via the `Runtime.cast$` and `Runtime.instanceof$` helper methods.

Eliminating method type parameters. A parametrized adapter class with an `apply` method is generated for each parametrized method, The adapter class is annotated with `@ParametricMethod`. The parametrized method is invoked by instantiating the adapter class through a generated factory method and invoking its `apply()` method.

Parametrized exceptions. Parametrized exceptions are treated just like other classes. Synthetic local classes, annotated with `@Instantiation`, are generated for each catch block with an instantiated generic exception class. Exception tables in the bytecode are rewritten with the new exception types.

Run-time instantiation. The `instanceof` and cast operations on constrained types or type variables are translated to similar operations on the instantiated type followed by calls to methods of the adapter object for the type that evaluate the constraint.

5. Discussion

Many features of modern object-oriented type systems can be cast as specializations of the generic constrained types framework. Generic constrained types generalize virtual types and have a connection to parametric types with use-site variance annotations, such as Java's wildcards.

5.1 Virtual types

Virtual types [27, 28, 13] are a language-based extensibility mechanism originally introduced in the language BETA [27] that—along with similar constructs built on path-dependent types found in languages such as Scala [41], J& [39], and Tribe [8]—share many similarities with type properties. A virtual type is a type binding nested within an enclosing instance. Subclasses are permitted to override the binding of inherited virtual types.

Virtual types may be used to provide genericity; indeed, one of the first proposals for genericity in Java was based on virtual types [50], and Java wildcards (i.e., parameters with use-site variance) were developed from a line of work beginning with virtual types [51, 22, 52].

Constrained types are more expressive than virtual types in that 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 [50] 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;
  T get(int i) { ... }
}
```

The virtual type `T` is unbound in `List`, but can be refined by binding `T` in a subclass:

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

Only classes where `T` is final bound, such as `IntList`, can be non-abstract. The analogous definition of `List` in X10 using type properties is as follows:

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

Unlike the virtual-type version, the X10 version of `List` is not abstract; `T` need not be instantiated by a subclass because it can be instantiated on a per-object basis. Rather than declaring subclasses of `List`, one uses the constrained subtypes `List{T<=Number}` and `List{T==Integer}`.

Type properties can also be refined contravariantly. For instance, one can write the type `List{T>=Integer}`, and even `List{Integer<=T, T<=Number}`.

5.2 Wildcards

As discussed in Section 2.4, type properties are similar to type parameters. Constrained types can provide a characterization of wildcards in Java [18, 52, 5]. Wildcards can be motivated by the following example from Torgersen et al. [52]: Consider a `Set` class and a variable `EMPTY` containing the empty set. What should be the type of `EMPTY`? In Java, one can use a wildcard and assign the type `Set<?>`, i.e., the type of all `Set` instantiated on *some* parameter. Clients of this type do not know what parameter to which the actual instance of `Set` is bound. With constrained types, a similar effect is achieved simply by leaving the element type property of `Set` unconstrained.

Wildcards can also be bounded above and below with "`? extends T`" and "`? super T`", respectively. A similar effect is achieved with constrained types by constraining the element type property of `Set` with a subtyping constraint.

We can define the following straightforward translation from Java wildcards to X10. Type parameters are translated to type properties whose name encodes their position in the parameter list. Types are translated as follows:

$$\begin{aligned}
[\![\texttt{X}]\!] &= \_i \text{ where } \texttt{X} \text{ is the } i\text{th type param}\\
[\![\texttt{C<?,}\ldots\texttt{>}]\!] &= \texttt{C\{}\ldots\texttt{\}}\\
[\![\texttt{C<? extends T,}\ldots\texttt{>}]\!] &= \texttt{C\{\_1 <= } [\![\texttt{T}]\!]\texttt{,}\ldots\texttt{\}}\\
[\![\texttt{C<? super T,}\ldots\texttt{>}]\!] &= \texttt{C\{\_1 >= } [\![\texttt{T}]\!]\texttt{,}\ldots\texttt{\}}\\
[\![\texttt{C<T,}\ldots\texttt{>}]\!] &= \texttt{C\{\_1 == } [\![\texttt{T}]\!]\texttt{,}\ldots\texttt{\}}
\end{aligned}$$

Through such a translation, the $\mathsf{FX}(\mathcal{G})$ calculus in Section 3 captures the essence of Java's wildcards, but extended with support for run-time type introspection. We leave to future work a formal proof of this characterization.

Like wildcards, constrained types support *proper abstraction* [52]. To illustrate, a `reverse` operation can operate on `List` of any type:

```
def reverse(list: List) {
  for (i: int in 0..list.length-1)
    swap(list, i, list.length-1-i);
}
```

The client of `reverse` need not provide the concrete type on which the list is instantiated; the `list` itself provides the element type—it is stored in the `list` to implement run-time type introspection.

In Java, this method would be written with a type parameter on the method; the type system permits it to be called with any `List`. However, the method parameter cannot be omitted: declaring a parameterless version of `reverse` requires delegating to a private parametrized version that "opens up" the parameter.

5.3 Structural constraints

Constraints on types need not be limited to subtyping. By introducing structural constraints on types, one can instantiate type properties on any type with a given set of methods or fields. A structural constraint is satisfied if the type has a member of the appropriate name and with a compatible type. This feature is useful for reusing code from separate libraries; it does not require code of one library to implement a named interface of the other library to interoperate.

Here, we consider an extension, not yet implemented, of the X10 type system that supports structural type constraints. Formally, the extension is straightforward; indeed the FX family already supports structural constraints via the rules for "$\texttt{x}\ \underline{\texttt{has}}\ \texttt{I}$" in Figure 6.

Structural constraints on types are found in many languages. For instance, Haskell supports type classes [23, 20]. In Modula-3, type equivalence is structural rather than nominal as in object-oriented languages of the C family (e.g., C++, Java, and X10). Unity [29] is another language with structural subtyping.

Using structural constraints in X10 was inspired by the language PolyJ [35], which allows type parameters to be bounded using structural *where clauses* [11]. For example, a sorted list class could be written as follows in PolyJ:

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

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

The analogous code for `SortedList` in the structural extension of X10 would be:

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

5.3.1 Function-typed properties

X10 supports first-class functions. Function-typed properties provide a useful alternative to structural constraints. In the following version of the `SortedList` class:

```
class SortedList(compare: (T,T)=>int)
    extends List {
  def add(x: T) {... compare(x, y) ...}
  ... }
```

the class has a property `compare` of type `(T,T)=>int`, a function that takes a pair of `T`s and returns an `int`.

Using this definition, one can create lists with distinct types using different constraints on the `compare` property. For example, one can define lists of both case-sensitive and case-insensitive strings:

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

`String.compareTo.(String)` selects the method named `compareTo` with a `String` argument from class `String`. The lists `unixFiles` and `windowsFiles` are constrained by different comparison functions. This allows the programmer to write code, for instance, in which it is illegal to pass a list of UNIX file names into a function that expects a list of Windows file names, and vice versa.

5.3.2 Optional methods

Structural method constraints permit the introduction of CLU-style optional methods [26]. Consider the following `Array` class:

```
class Array[T] {
  def add(a: Array[S])
    {T has add(S): U}: Array[U] {
    return new Array[U](
      (p:Point) => this(p)+a(p));
  }
  ... }
```

The `Array` class defines an `add` method that takes an array of S, adds each element of the array to the corresponding element of `this`, and returns an array of the results. The method constraint specifies that the method may only be invoked if `T` has an `add` method of the appropriate type. Thus, for example, an `Array[int]` can be added to an `Array[double]` because `int` has a method `add` (corresponding to the + operation) that adds an `int` and a `double`, retuning a `double`. However, `Array[Rabbit]`, for example, does not support the `add` operation because `Rabbit` does not have an `add` method.

5.4 Run-time casts

While constraints are normally solved at compile time, constraints can be evaluated at run time by using casts. The expression `xs as List{length==n}` checks not only that `xs` is an instance of the `List` class, but also that `xs.length` equals `n`. A `ClassCastException` is thrown if the check fails. In this example, the test of the constraint does not require run-time constraint solving; the constraint can be checked by simply evaluating the `length` property of `xs` and comparing against `n`. However, the situation is more complicated when casting to a type that constrains the value's type properties.

For equality constraints on types, one can simply test equality on canonical run-time representations of the base types and of the constraints. However, with subtyping constraints (e.g., S<=T), the generated code must be able to test if the constraint on the subtype entails the constraint on the supertype. Since one or both of the types involved in the constraint might be type variables, the entailment cannot be checked at compile time.

One approach is to restrict the language to rule out casts to type parameters and to generic types with subtyping constraints, ensuring that entailment checks are not needed at run time.

Alternatively, the constraint solver could be embedded into the runtime system. This is the solution used in the X10 implementation; however, this solution can result in inefficient run-time casts if entailment checking for the given constraint system is expensive.

Another option is to test objects cast to `T` not for membership in the type `T`, but rather to test against the *interpretation* of `T`. Observe that if an instance of a generic class `C[X]` is a member of the type `C{X==U}`, then all fields $\mathtt{f}_i$ of the instance with declared type S_i contain values that are instances of S_i[U/X]. For example, given the following declaration of class `List`:

```
class List[X] {
  val head: X;
  val tail: List[X];
}
```

if `xs` is an instance of `List{X==String}`, then by checking that `xs.head` is an instance of `String` and `xs.tail` is, recursively, an instance of `List{X==String}`. This property can be exploited by implementing cast to check the types of all fields of the object. For this check to be sound, it is vital that all fields whose type depends on the type property `X` be transitively final; otherwise, the test is not invariant—the result of the test could change as the data is mutated. Care must also be taken to implement the test so that it terminates for cyclic data structures. This implementation is inefficient for large data structures.

This solution has a more permissive semantics than those implemented in X10 or FX($\mathcal{G}$,$\mathcal{A}$). The difference is best illustrated by considering an empty generic class:

```
class Nil[X] { }
```

In this case, there is no field of type `X` to test; therefore, an object instantiated as `Nil[int]` can be considered a member of `Nil[String]`. However, the solution remains sound: Given a class `C[X]` and an expression `e` of type `t.X`, if a run-time check finds that `t` has type `C[T]`, the compiler *cannot* use this information to derive that `e` has type `T`. We leave to future work a proof of this claim.

6. Related work

Constraint-based type systems, dependent types, and generic types have been well studied in the literature. Further discussion of related work for constrained types can be found in our earlier work [40].

Constraint-based type systems. The use of type constraints for type inference and subtyping was first proposed by Mitchell [34] and Reynolds [43]. HM(X) [48] is a constraint-based framework for Hindley–Milner-style type systems. The framework is parametrized on the specific constraint system X; instantiating X yields extensions of the HM type system. Constraints in HM(X) are over types, not values. The HM(X) approach is an important precursor to our constrained types approach. The principal difference is that HM(X) applies to functional languages and does not integrate dependent types. We consider object-oriented languages with constraint-based type systems when we discuss generic types, below.

Dependent types. Dependent type systems [55, 31, 2] parametrize types on values. Our work is closely related to Dependent ML (DML [55]), which is also built parametrically on a constraint solver. The main distinction between DML and constrained types lies in the target domain: DML is a functional programming language; constrained types are designed for imperative, concurrent object-oriented languages. Types in DML are refinement types [17]: they do not affect the operational semantics, and erasing the constraints yields a legal DML program. This differs from generic constrained types, where erasure of subtyping constraints can prevent the program from type-checking. DML does not permit any run-time checking of constraints (dynamic casts). Another distinction between DML and constrained types is that constraints in DML are defined over a set of "index" variables are introduced; in X10, constraints are defined over program variables and types.

Logically qualified types, or liquid types [44], permit types in a base Hindley–Milner-style type system to be refined with conjunctions of logical qualifiers. The subtyping relation is similar to X10's; that is, two liquid types are in the subtyping relation if their base types are in the relation and if one type's qualifier implies the other's. Liquid types support type inference and the type system is path sensitive; neither is the case in X10. Liquid types do not provide subtyping constraints.

Another dependent-type approach is to integrate theorem provers into the programming language. Concoqtion [16] extends types in OCaml [25] with constraints written as Coq [9] rules. While the types are expressive, supporting the full generality of the Coq language, proofs must be provided to satisfy the type checker. X10 supports only constraints that can be checked by a constraint solver during compilation. Ynot [36] is an extension to Coq for reasoning about dependently-typed functional programs with side-effects using a form of separation logic. Constrained types permit a more natural expression of constraints than Coq formulas; however, unlike Ynot, they do not capture side-effects since they constrain only mutable object state.

Genericity. Genericity in object-oriented languages is usually supported through type parametrization.

A number of proposals for adding genericity to Java quickly followed the initial release of the language [4, 35, 50, 1]. GJ [4] implements invariant type parameters via type erasure. PolyJ [35] supports run-time representation of types via adapter objects, and also permits instantiation of parameters on primitive types and structural parameter bounds. Viroli and Natali [54, 53] also support a run-time representation of types, using Java's reflection API. NextGen [6, 1] was implemented using run-time instantiation. X10's generics have a hybrid implementation, adopting PolyJ's adapter object approach for dependent types and for type introspection and using NextGen's run-time instantiation approach for greater efficiency.

C$^\sharp$ also supports generic types via run-time instantiation in the CLR [49]. Type parameters may be declared with definition-site variance tags. Generalized type constraints were proposed for C$^\sharp$ [12]: methods can be annotated with subtyping constraints that must be satisfied to invoke the method. Generic X10 supports these constraints, as well as constraints on values, with method and constructor where clauses.

Contracts. Constrained types have many similarities to contracts [42, 32, 14], constraint annotations on methods and classes. A key difference is that contracts are dynamically enforced assertions; whereas, constrained types are statically enforced. Dynamic enforcement enables a more expressive assertion language at the cost of possible run-time failures. Integration of constrained types and generic types effectively allow parametrization of classes on constraints. Consider the `List` class in Figure 1. If the class is instantiated on a constrained type `C{c}`, i.e., `List[C{c}]`, then all elements of the list satisfy `c`. Guha et al. [19] explore polymorphic contracts as a way to achieve this parametrization in a functional language. Generic constrained types provide constraint parametrization in a natural extension of object-oriented languages.

ESC/Java [15], Spec# [3], and JML [24] are instances of static assertion checkers for OO languages. Constrained types offer a less expressive, but more modular, lightweight solution.

7. Conclusions

We have presented a constraint-based framework FX for type- and value-dependent types in an object-oriented language. The use of constraints on type properties allows the design to capture many features of generics in object-oriented languages and then to extend these features with more expressive power. We have proved the type system sound.

The type system FX($\mathcal{G}, \mathcal{A}$) formalizes the semantics of the X10 programming language. The design admits an efficient implementation for generics and dependent types in X10. To improve the expressiveness of X10, we plan to implement a type inference algorithm that infers constraints over types and values, and to support user-defined constraints.

References

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

[2] Lennart Augustsson. Cayenne: a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 239–250, 1998.

[3] Mark Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec♯ programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices, International Workshop, CASSIS 2004*, volume 2263 of *LNCS*. Springer-Verlag, 2004.

[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] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP)*, number 5142 in Lecture Notes in Computer Science, pages 2–26, July 2008.

[6] Robert Cartwright and Guy L. Steele. Compatible genericity with run-time types for the Java programming language. In *Proc. OOPSLA '98*, Vancouver, Canada, October 1998.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[8] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 121–134, 2007.

[9] The Coq proof assistant: Reference manual, version 8.1. `http://coq.inria.fr/`, 2006.

[10] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.

[11] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proc. OOPSLA '95*, pages 156–168, Austin TX, October 1995. ACM SIGPLAN Notices 30(10).

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

[13] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proc. 33th ACM Symp. on Principles of Programming Languages (POPL)*, pages 270–282, Charleston, South Carolina, January 2006.

[14] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, October 2002.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, Berlin, Germany, June 2002.

[16] Seth Fogarty, Emir Pašalić, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 112–121, January 2007.

[17] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, June 1991.

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

[19] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium*, 2007.

[20] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[21] 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.

[22] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(5):795–847, 2006.

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

[24] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, Minneapolis, Minnesota, 2000.

[25] Xavier Leroy et al. The Objective Caml system. `http://caml.inria.fr/ocaml/`.

[26] Barbara Liskov et al. *CLU Reference Manual*. Springer-Verlag, 1984.

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

[28] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Proc. OOPSLA '89*, pages 397–406, October 1989.

[29] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP)*, number 5142 in Lecture Notes in Computer Science, July 2008.

[30] Per Martin-Löf. *A Theory of Types*. 1971.

[31] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[32] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[33] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[34] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL'84)*, pages 174–185, 1984.

[35] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[36] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.

[37] Karl A. Nyberg, editor. *The annotated Ada reference manual*. Grebyn Corporation, Vienna, VA, USA, 1989.

[38] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 138–152, Warsaw, Poland, April 2003.

[39] 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.

[40] 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.

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

[42] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[43] John C. Reynolds. Three approaches to type structure. In *Proceedings of TAPSOFT/CAAP 1985*, volume 185 of *LNCS*, pages 97–138. Springer-Verlag, 1985.

[44] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

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

[46] Vijay Saraswat et al. The X10 language specification. Technical report, IBM T.J. Watson Research Center, 2008.

[47] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR*, 2005.

[48] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.

[49] Don Syme and Andrew Kennedy. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[50] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP '97*, number 1241 in Lecture Notes in Computer Science, pages 444–471, 1997.

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

[52] 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.

[53] Mirko Viroli. A type-passing approach for the implementation of parametric methods in java. *The Computer Journal*, 46(3):263–294, 2003.

[54] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Proc. OOPSLA '00*, pages 146–165, 2000.

[55] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227, San Antonio, TX, January 1999.