# Genericity through Constrained Types


Nathaniel Nystrom [*]
nystrom@us.ibm.com

Igor Peshansky [*]
igorp@us.ibm.com

Vijay Saraswat [*]
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 OO languages can be viewed as constrained types.

This paper extends the constrained types approach to handle *type-dependency* ("genericity"). The key idea is to formalize the essence of nominal object-oriented types itself as a constraint system over predicates such as `X` extends `T`, `S` is a subtype of `T` and `X` has member `I`. 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 $\mathcal{C}$, we obtain languages with the power of FJ, FGJ, dependent-types, and new OO languages which 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, self types, and Scala's path-dependent types.


## 1. Introduction

Modern architectural advances are leading to the development of complex computational systems, such as heterogeneous multi-core systems (e.g. Cell), large CPU count systems such as the Blue Gene, and hybrid accelerated clusters such as the Road Runner. Such systems pose a central challenge: How can application programmers write high-performance programs for such machines while building on the productivity gains of modern OO languages?

The X10 programming language [53, 10, 52] was designed to address the challenges of "productivity with performance" on these diverse architectures. Developed on the *asynchronous partitioned global address* (APGAS) model, X10 organizes computation into a collection of logical *places*, which encpasulates data and one or more *activities* that operate on the data. Places capture the idea of *locality* and *heterogenity*. Data resides in a global address space; thus a field of an object can point to an object in a different place. Remote data is not operated on directly; rather light-weight activities must be spawned remotely to operate on them.

The design of X10 requires the development of a rich type system to enable code reuse, permit a large variety of errors to be caught at compile-time, and to generate efficient code. For instance, a central data structure in X10 is the dense, distributed, multi-dimensional array. Arrays are defined over a set of indices known as *regions*, may support arbitrary base types, and accesses through *points* that must lie in the underlying region. For performance it is necessary that as far as possible array index accesses are bounds-checked statically. Further, certain regions (such as rectangular regions) may be represented particularly efficiently. Hence if a variable is to range only over rectangular 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 will ensure that no race conditions arise due to simultaneous accesses by multiple activities to different pieces.

These requirements motivated us to develop a framework for dependent types in OO languages [50]. *Dependent type systems* [36, 63, 44, 5, 6, 4, 13] 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*. Constraint systems were originally developed in [51] 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 OO 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 may 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 type `C{c}` where `c` is a constraint on the immutable fields of the class. 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. 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.


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


[1] The use of constraints for types has a distinguished history going back to Mitchell [39]. Our work is closely related to the HM($X$) approach [54] – see Section 5.4 for details.

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(X)[63]. 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 availability of fields and methods (conditional fields and methods). Final variables in the computation can be used directly in types; there is no need for a separate parallel universe of index expressions 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).

### 1.1 Generic types

In this paper we extend the constrained types approach to handle *generic types* [33, 41, 7, 40, 57, 22, 56] – 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 (public) 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 is a subtype of `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` $\mathtt{m}(\overline{\mathtt{x}}{:}\overline{S}){:}\mathtt{T}$ to express this.

The entailment relation between these pieces of partial information is straightforward to specify, given the interpretation specified in the previous paragraph. For instance, the `<:` relation on class names must be transitive and is precisely given by the transitive closure of the extends relation. Section 4.1 describes the resulting Object constraint system, $\mathcal{O}$, in more detail.

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 [22], Scala [43] and in functional languages such as ML [38] and Haskell [30]). 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 that classes have type *parameters* [22], the constrained type approach suggests an alternative: classes may have type-valued properties (=immutable instance fields). To create an instance of such a class a type must be supplied to initialize this property. As with other state of the object, 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- and type-dependency of types.

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

EXAMPLE 1.1 (Array). *Consider the class* `Array` *declared as:*

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

*The class has two properties, a type-valued property* `T` *(enclosed in square brackets), and a value property* `r` *of type* `Region`*. The* `get` *method for the array requires a point* `p` *which must lie in the object's region.*[3]

The type `Array{self.T==int}` specifies the type of all arrays whose base type (`T`) is `int`. `Array{self.T`$\trianglelefteq$ `Numeric, self.r.rank==2}` specifies the type of all arrays whose base type is a subtype of `Numeric`, and whose region has rank `2`. The type `Array{self.T==List{self.length==N}, self.r.rank==N}` specifies an `N`-dimensional array whose elements are lists (of unkown 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 (e.g. `String` or `Point{self in R}`. Thus if `a` is a "final" access path (e.g. `x.f.g`, where `x` is an immutable variable, and `f` and `g` are immutable fields), the expression `a.T` may itself be used as a type (cf. the return type `this.T` for `Array.get`).

### 1.2 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 a new revision of the X10 language (v1.7, to be released Fall 2008), and various programming idioms. Besides constrained types, the language supports function types. Since there may be a large number of constrained types in a program, a pure heterogeneous translation may lead to significant code bloat. Instead we use a hybrid implementation scheme that combines ideas from NextGen [9, 2, 3] and PolyJ [40]. 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 primitive values be boxed.

We also discuss relationship with virtual types, Java wildcards, self types and structural constraints.

### 1.3 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[25]), with constrained types and constraints on classes, fields and methods. A single set of rules specifies the static and dynamic semantics for all languages in the family. The static semantics is shown to 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 constraints from the Object constraint system $\mathcal{O}$ described above – it corresponds to FGJ (being somewhat richer in permitting path types). FX($\mathcal{A}$) permits the use of constraints from

[3] 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 replaced by `r` and `T` respectively.

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

**Figure 1.** List example, simplified

some constraint system $\mathcal{A}$ – it corresponds to a language with a pure value-dependent type system as described in [50]. $\mathsf{FX}(\mathcal{G},\mathcal{A})$ permits the use of constraints from $O$ and $\mathcal{A}$ – thus permitting value- and type-dependence – and is the main topic for 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, $\mathsf{FX}(\mathcal{C})$ 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. The implementation of generics in X10 by translation to Java is described in Section 3. Section 4 presents a formal semantics and a proof of soundness. Section 5 discusses extensions of the type system, including extensions for virtual types and self types, and related work. Finally, Section 6 concludes.

## 2. X10 language overview

This section presents an informal description of dependent and generic types in X10.

X10 is a class-based object-oriented language. The language has a sequential core similar to Java or Scala, but 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{e}`, where `C` is the name of a class and `e` is a constraint on the properties of `C` and the final variables in scope at the type. The constraint `e` 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 `length` is declared as in Figure 1. Classes in X10 may be declared with any number of type properties and value properties.

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

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

  def get(i: int{0 <= i, i < length}){length > 0} = {
    return i==0 ? head : tail.get(i-1);
  }

  def map[S](f: Object{self :> T} => S): List[S] = {
    if (length==0)
      return new List[S](0);
    else if (length==1)
      return new List[S](f(head));
    else
      return new List[S](f(head), tail.map[S](f));
  }

  def this[S](): List[S](0) = property[S](0);
  def this[S](hd: S): List[S](1) = {
    property[S](1); head = hd;
  }
  def this[S](hd: S, tl: List[S]): List[S](tl.length+1) = {
    property[S](tl.length+1);
    head = hd; tail = tl;
  }
}
```

**Figure 2.** List example, with more constraints

`this` occurring in the constraint refers to the instance of the enclosing `List` class, 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](tl.length+1)`, that is, `List{self.T==S, self.length==tl.length+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 present a version of `List` where we write invariants to be enforced statically. Consider the new version in Figure 2.

### 2.1 Class invariants

Properties of a class may be constrained with a *class invariant*. The `List` declaration's class invariant specifies that the length of the list be 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 instance, a binary tree class might require that its elements implement the `Comparable` interface:

```
class Tree[T]{T <: Comparable[T]} {
  left, right: Tree[T]; ...
}
```

### 2.2 Class member invariants

Class and interface member declarations may have additional constraints that must be satisfied for access.

The field declarations in Figure 2 each have a *field constraint*. The field constraint on `head` requires that `this.length>0`; that is `this.head` may not be dereferenced unless `this` has type `List{length>0}`. Similarly, `tail` cannot be accessed unless the list has a non-empty tail. The compiler is free to generate optimized representations of instances of `List` with a given length: it may remove the the `head` and `tail` fields for empty lists, for instance. Similarly, the compiler may specialize instances of `List` with a given concrete type for `T`. This specialization is described in Section 3.

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 *method constraint* that requires that the actual receiver's `length` field must be non-zero—calls to `get` on empty lists are not permitted. A method with a method constraint is called a *conditional method*. The constraint on `get` ensures that the field constraint 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`. Note that for this example to type-check, the constraint system must establish the field constraint on `tail` and the method constraint on the recursive call; that is, it must be able guarantee that the `tail` is non-empty and that `i-1` is within the bounds of `tail`.

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 constraint may be weakened by an overriding method; that is, the method constraint in the superclass must entail the method constraint 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 a supertype of `T` to `S`. A parametrized method is invoked by giving type arguments before the expression arguments (see recursive call to `map`).[4]

`List` also defines three constructors: the first constructor takes no value arguments and initializes the length to `0`. Note that `head` and `tail` are not assigned since they are inaccessible. The second constructor takes an argument for the head of the list; the third takes both a head and tail.

### 2.3 Type constraints and variance

Use-site variance based on structural virtual types were proposed by Thorup and Torgerson [58] and extended for parametrized type systems by Igarashi and Viroli [26]. The latter type system lead to the development of wildcards in Java [22, 59, 8].

Type properties and subtyping constraints may be used in X10 to provide use-site variance constraints.

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 l`, the return type of the `get` method is `l.T`. Since the property `T` is unconstrained, the caller can only assign the return value of `get` to a variable of type `l.T` or of type `Object`.

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

- `List{T==float}`. The type property `T` is bound to `float`. For a final expression `l` of this type, `l.T` and `float` are equivalent types and can be used interchangeable wherever `l` is in scope.
- `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 `l` has the type `List{T<:Collection}`, then the return type of `get` has type `l.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 l` of this type, any supertype of `String` may flow into a variable of type `l.T`. The return type of the `get` method is known to be a supertype of `String` (and implicitly a subtype of `Object`).

In the shortened syntax for types (e.g., `List[T](n)`), an actual type argument `T` may optionally be annotated with a *use-site variance tag*, either + or -: if `X` is a type property, 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}`.

## 3. Translation

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

The design is a hybrid design combining techniques of run-time instantiation from NextGen [9, 2, 3] and type-passing from PolyJ [40]. Generic classes are translated into "template" classes that are instantiated on demand at run time by binding the type properties to concrete types. Value constraints 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 framework, which translates X10 source to Java source.[5] The X10 runtime was 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 constraints on types, not values; we leave value-constraint specialization to future work. For example, consider the following classes.

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

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

The compiler generates the following code:

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

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

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

The member class `A.T` is used in place of the type property `T`. The field `T` 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 runtime 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;
}
```

The class loader rewrites allocations of template classes (e.g., `A`) with allocations of the instantiated classes (e.g., `A$$int`). The name of the instantiated class is a mangled name derived from the name of the template class and the type properties. This code cannot be generated directly because class `A$$int` does not yet exist; the Java source compiler would fail to compile `C`.

Upon evaluation of the constructor, 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 `T` with the concrete type `int`.

Parameter types are coerced to and from the actual type `X` (a Java primitive type or `Object`) using methods `Runtime.to$X(Object)` and `Runtime.from(X)`, 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. `@InstantiateClass` annotates 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`. `@Instantiation` is used for parametrized exceptions.

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.*** For each parametrized method, we generate a parametrized adapter class annotated with `@ParametricMethod` and a factory method. A parametrized method is invoked by calling the factory method and invoking its `apply()` method on the resulting adapter object.

***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.*** We translate `instanceof` and cast operations on constrained types or type variables to similar operations on the instantiated type followed by calls to methods of the adapter object for the type that evaluate the constraint.

# 4. Semantics

We now describe the semantics of languages in the FX family. For uniformity we declare type-valued parameters and properties to be of "type" `type`, instead of using square brackets to demarcate them.

Each language $\mathcal{L}$ in the family is defined over a given input constraint system $\mathcal{X}$. Given a program $P$, we now show how to build a larger constraint system $\mathcal{O}(\mathcal{X})$ on top of $\mathcal{X}$ which captures constraints related to the object-oriented structure of $P$. $\mathcal{O}$ is sensitive to $\mathcal{X}$ only in that $\mathcal{O}$ depends on the types defined by $\mathcal{L}$, and these may depend on $\mathcal{X}$. For some members of $\mathcal{L}$, viz. the generic languages, $\mathcal{X}$ itself may use some of the constraints defined by $\mathcal{O}$. Thus we should think of $\mathcal{X}$ and $\mathcal{O}$ as being defined simultaneously and recursively.

The static and dynamic semantics of $\mathcal{L}$ uses $\mathcal{O}(\mathcal{X})$.

## 4.1 The Object constraint system, $\mathcal{O}$

The inference relation for $\mathcal{O}$ depends on the object-oriented structure of the input program $P$ in $\mathcal{L}$.

The constraints of $\mathcal{O}$ are given by:

$$\begin{array}{rlll}
\text{(Member)} & \mathtt{I} & ::= & \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{V}})\{\mathtt{c}\}:\mathtt{T}=\mathtt{e} \mid \mathtt{f}{:}\mathtt{V} \\
\text{(Const.)} & \mathtt{c,d} & ::= & \mathtt{class(C)} \mid \mathtt{S} \trianglelefteq \mathtt{T} \mid \mathtt{S <: T} \\
 & & & \mid \mathtt{fields(x)} = \overline{\mathtt{f}}:\overline{\mathtt{V}} \\
 & & & \mid \mathtt{x\ has\ I}
\end{array}$$

`class(C)` is intended to be true for all classes `C` defined in the program. $\mathtt{S} \trianglelefteq \mathtt{T}$ is intended to hold if it can be established that `S` extends `T`, for instance if `S` is a class that extends `T`, or if `S` is a type variable and `T` its upper bound. `S <: T` is intended to hold if `S` is a subtype of `T`. For a variable `x`, `fields(x)` is intended to specify the (complete) set of typed fields available to `x`. `x has I` is intended to specify that the member `I` (field or method) is available to `x`—for instance it is defined at the class at which `x` is declared or inherited by it, or it is available at the upper bound of a type variable.

$\mathcal{O}$ satisfies the axioms and inference rules in Figure 3. Since $\mathcal{O}$ is a constraint system [51], it also satisfies Identity and Cut.

Note that some rules (viz, S-EXISTS rules) use $\vdash$.

We assume that the rules given are complete for defining the predicates `C<:D` and `C has I`, for classes `C`, `D` and members `I`; that is, if the rules cannot be used to establish $\vdash_{\mathcal{O}} \mathtt{C <: D}$ ($\vdash_{\mathcal{O}} \mathtt{C\ has\ I}$), then it is the case that $\vdash_{\mathcal{O}} \mathtt{C} \not<: \mathtt{D}$ ($\vdash_{\mathcal{O}} \neg(\mathtt{C\ has\ I})$).

Such negative facts are important to establish *inconsistency* of assumptions (for instance, for the programming languages which permits the user to state constraints on type variables).

## 4.2 Judgments

In the following $\Gamma$ is a *well-typed context*, i.e. a (finite, possibly empty) sequence of formulas `x : T`, `T type` and constraints `c` satisfying:

1. for any formula $\phi$ in the sequence all variables `x` (`X`) occurring in $\phi$ are defined by a declaration `x : T` (`X type`) in the sequence to the left of $\phi$.
2. for any variable `x` (`X`), there is at most one formula `x : T` (`X type`) in $\Gamma$.

The judgments of interest are as follows. (1) Type well-formedness: $\Gamma \vdash \mathtt{T\ type}$, (2) Subtyping: $\Gamma \vdash \mathtt{S <: T}$, (3) Typing: $\Gamma \vdash \mathtt{e : T}$, (4) Method OK (method `M` is well-defined for the class `C`): $\Gamma \vdash M$ OK in $C$, (5) Field OK (field `f : T` is well-defined for the class `C`): $\Gamma \vdash f : T$ OK in $C$ (6) Class OK: $\Gamma \vdash \mathtt{L}$ *OK* (class definition `L` is well-formed).

In defining these judgments we will use $\Gamma \vdash_{\mathcal{C}} c$, the judgment corresponding to the underlying constraint system. For simplicity, we define $\Gamma \vdash \mathtt{c}$ to mean $\sigma(\Gamma) \vdash_{\mathcal{C}} \mathtt{c}$, where the *constraint projection*, $\sigma(\Gamma)$ is defined as allows.

$$\frac{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{V}})\ \mathtt{extends}\ \mathtt{D}\ \ldots \in \mathtt{P}}{\vdash_{O} \mathtt{class}(\mathtt{C}), \mathtt{C} \trianglelefteq \mathtt{D}}\ (\textsc{Class}) \qquad \frac{\Gamma \vdash_{O} \mathtt{fields}(\mathtt{x}) = \overline{\mathtt{f}}:\overline{\mathtt{V}}}{\Gamma \vdash_{O} \mathtt{x}\ \mathrm{has}\ \mathtt{f_i}:\mathtt{V_i}}\ (\textsc{Has-F}) \qquad \vdash_{O} \mathtt{new}\ \mathtt{D}(\overline{\mathtt{t}}).\mathtt{f_i} = \mathtt{t_i}\ \ (\textsc{Sel}) \qquad \frac{\Gamma \vdash_{O} \mathtt{x}:\mathtt{C}, \mathtt{class}(\mathtt{C})}{\Gamma \vdash_{O} \mathit{inv}(\mathtt{C}, \mathtt{x})}\ (\textsc{Inv})$$

$$\vdash_{O} T \trianglelefteq T \quad (\textsc{V-Id}) \qquad \frac{\Gamma \vdash_{O} \mathtt{t} \trianglelefteq \mathtt{T}}{\Gamma \vdash_{O} \mathtt{t} <: \mathtt{T}}\ (\textsc{Sub-X}) \qquad \frac{\Gamma \vdash_{O} \mathtt{T_1} <: \mathtt{T_2}, \mathtt{T_2} <: \mathtt{T_3}}{\Gamma \vdash_{O} \mathtt{T_1} <: \mathtt{T_3}}\ (\textsc{S-Trans})$$

$$\frac{\Gamma, \mathtt{c} \vdash_{O} \mathtt{S} <: \mathtt{T}}{\Gamma \vdash_{O} \mathtt{S}\{\mathtt{c}\} <: \mathtt{T}}\ (\textsc{S-Const-L}) \qquad \frac{\Gamma \vdash_{O} \mathtt{S} <: \mathtt{T} \qquad \Gamma, \mathtt{self}:\mathtt{S} \vdash_{O} \mathtt{c}}{\Gamma \vdash_{O} \mathtt{S} <: \mathtt{T}\{\mathtt{c}\}}\ (\textsc{S-Const-R}) \qquad \frac{\mathtt{class}\ \mathtt{C}(\ldots)\ \mathtt{extends}\ \mathtt{D}\{\ldots\} \in \mathtt{P}}{\vdash_{O} \mathtt{C} <: \mathtt{D}}\ (\textsc{S-Extends})$$

$$\frac{\Gamma \vdash \mathtt{U}\ \mathtt{type} \qquad \Gamma \vdash_{O} \mathtt{S} <: \mathtt{T} \qquad (\mathtt{x}\ \text{fresh})}{\Gamma \vdash_{O} \mathtt{x}:\mathtt{U};\mathtt{S} <: \mathtt{T}}\ (\textsc{S-Exists-L}) \qquad \frac{\Gamma \vdash \mathtt{t}:\mathtt{U} \qquad \Gamma \vdash_{O} \mathtt{S} <: \mathtt{T}[\mathtt{t}/\mathtt{x}]}{\Gamma \vdash_{O} \mathtt{S} <: \mathtt{x};\mathtt{U}:\mathtt{T}}\ (\textsc{S-Exists-R}) \qquad \frac{\Gamma \vdash_{O} \mathtt{S} <: \mathtt{T} \qquad \Gamma \vdash_{O} \mathtt{T} <: \mathtt{S}}{\Gamma \vdash_{O} \mathtt{S} \equiv \mathtt{T}}\ (\textsc{Type-equiv})$$

$$\mathtt{x}:\mathtt{Object} \vdash_{O} \mathtt{fields}(\mathtt{x}) = \bullet\ (\textsc{Fields-B}) \qquad \frac{\begin{array}{c}\Gamma, \mathtt{x}:\mathtt{D} \vdash_{O} \mathtt{fields}(\mathtt{x}) = \overline{\mathtt{g}}:\overline{\mathtt{V}} \\ \mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{U}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{D}\{\overline{\mathtt{M}}\} \in \mathtt{C}\end{array}}{\Gamma, \mathtt{x}:\mathtt{C} \vdash \mathtt{fields}(\mathtt{x}) = \overline{\mathtt{g}}:\overline{\mathtt{V}}, \overline{\mathtt{f}}:\overline{\mathtt{U}[\mathtt{x}/\mathtt{this}]}}\ (\textsc{Fields-I}) \qquad \frac{\Gamma, \mathtt{x}:\mathtt{S} \vdash_{O} \mathtt{fields}(\mathtt{x}) = \overline{\mathtt{f}}:\overline{\mathtt{V}}}{\begin{array}{c}\Gamma, \mathtt{x}:\mathtt{S}\{\mathtt{d}\} \vdash_{O} \mathtt{fields}(\mathtt{x}) = \overline{\mathtt{f}}:\overline{\mathtt{V}\{\mathtt{d}[\mathtt{x}/\mathtt{self}]\}} \\ \Gamma, \mathtt{x}:(\mathtt{y}:\mathtt{U};\mathtt{S}) \vdash_{O} \mathtt{fields}(\mathtt{x}) = \overline{\mathtt{f}}:\overline{(\mathtt{y}:\mathtt{U};\mathtt{V})}\end{array}}\ (\textsc{Fields-C,E})$$

$$\frac{\begin{array}{c}\Gamma, \mathtt{x}:\mathtt{C} \vdash_{O} \mathtt{class}(\mathtt{C}) \qquad \theta = [\mathtt{x}/\mathtt{this}] \\ \mathtt{def}\ \mathtt{m}(\overline{\mathtt{z}}:\overline{\mathtt{V}})\{\mathtt{c}\}:\mathtt{T} = \mathtt{e} \in P\end{array}}{\Gamma, \mathtt{x}:\mathtt{C} \vdash_{O} \mathtt{x}\ \mathrm{has}\ (\mathtt{m}(\overline{\mathtt{z}}:\overline{\mathtt{V}\theta})\{\mathtt{c}\theta\}:\mathtt{T}\theta = \mathtt{e})}\ (\textsc{Method-B}) \qquad \frac{\begin{array}{c}\Gamma, \mathtt{x}:\mathtt{D} \vdash_{O} \mathtt{x}\ \mathrm{has}\ \mathtt{m}(\overline{\mathtt{z}}:\overline{\mathtt{V}})\{\mathtt{c}\}:\mathtt{T} = \mathtt{e} \\ \mathtt{class}\ \mathtt{C}(\ldots)\ \mathtt{extends}\ \mathtt{D}\{\overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}\end{array}}{\Gamma, \mathtt{x}:\mathtt{C} \vdash_{O} \mathtt{x}\ \mathrm{has}\ \mathtt{m}(\overline{\mathtt{z}}:\overline{\mathtt{V}})\{\mathtt{c}\}:\mathtt{T} = \mathtt{e}}\ (\textsc{Method-I}) \qquad \frac{\Gamma, \mathtt{x}:\mathtt{S} \vdash_{O} \mathtt{x}\ \mathrm{has}\ \mathtt{m}(\overline{\mathtt{z}}:\overline{\mathtt{V}})\{\mathtt{c}\}:\mathtt{T} = \mathtt{e}}{\begin{array}{c}\Gamma, \mathtt{x}:\mathtt{S}\{\mathtt{d}\} \vdash_{O} \mathtt{x}\ \mathrm{has}\ \mathtt{m}(\overline{\mathtt{z}}:\overline{\mathtt{V}})\{\mathtt{c}\}:\mathtt{T}\{\mathtt{d}[\mathtt{x}/\mathtt{self}]\} = \mathtt{e} \\ \Gamma, \mathtt{x}:(\mathtt{y}:\mathtt{U};\mathtt{S}) \vdash_{O} \mathtt{x}\ \mathrm{has}\ \mathtt{m}(\overline{\mathtt{z}}:\overline{\mathtt{V}})\{\mathtt{c}\}:(\mathtt{y}:\mathtt{U};\mathtt{T}) = \mathtt{e}\end{array}}\ (\textsc{Method-C,E})$$

For a class C and variable x, $\mathit{inv}(\mathtt{C}, \mathtt{x})$ stands for the conjunction of class invariants for C and its supertypes, with `this` replaced by x.

**Figure 3.** The Object constraint system, $O$

$$\begin{array}{l}\sigma(\varepsilon) = \mathtt{true} \\ \sigma(\mathtt{x}:\mathtt{T}\{\mathtt{c}\}, \Gamma) = \mathtt{c}[\mathtt{x}/\mathtt{self}], \sigma(\Gamma) \\ \sigma(\mathtt{x}:(\mathtt{y}:\mathtt{S};\mathtt{T}), \Gamma) = \sigma(\mathtt{y}:\mathtt{S}, \mathtt{x}:\mathtt{T}, \Gamma) \\ \sigma(c, \Gamma) = \mathtt{c}, \sigma(\Gamma)\end{array}$$

Above, in the third rule we assume that alpha-equivalence is used to choose the variable x from a set of variables that does not occur in the context $\Gamma$.

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

### 4.3 FX(·)

The semantics of FX(·) is presented in Figure 4.

The syntax is essentially that of FJ with three major exceptions. First, types may be constrained with a clause {c}. Second 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 clause, and in the case of methods, c is an additional condition that must be satisfied by `this` and the arguments of the method in order for the method to be applicable.

Note that we distinguish the category of *parameter types* (V) from types (T). This is in preparation for the introduction of type variables; we will introduce a "type" `type` and permit parameters and fields to have this type, thus supporting genericity.[6]

The syntax for constraints in FX(·) is specified in Figure 4. 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 4 that are not user constraints are necessary to define the static and dynamic semantics of the program (see, e.g. the rules T-New and T-Field, T-Var that use equalities, conjunctions and existential quantifications. The use of this richer constraint set is not necessary, it simply enables us to present the static and dynamic semantics once for the entire family of FX languages, distinguishing different members of the family by varying the constraint system over which they are defined.[7]

The set of types includes classes C and is closed under constrained types (T{c}) and existential quantification (`x:S;T`). An object o is of type C (for C a class) if it is an instance of a subtype of C; it is of type T{c} if it is of type T and it satisfies the constraint c[o/self][8]; it is of type `x:S;T` if there is some object q of type S such that o is of type `T[q/x]` (treating at type as a syntactic expression).

The rules for well-formedness of types are straightforward, given the assumption that constraints are of a pre-given type $o$.

***Type judgment rules.*** T-Var is as expected, except that it asserts the constraint `self==x` which records the fact 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

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

[7] Another way of saying this is that the language $\mathcal{L} = \mathsf{FX}(\mathcal{C})$ is actually defined over the constraint system $O(\mathcal{C})$–the constraints of $O$ are uniformly added to every FX member.

[8] Thus the constraint c in a type T{c} should be thought of as a unary predicate λ`self.c`, an object is of this type if it is of type T and satisfies this predicate.

**FX productions:**

$$
\begin{array}{rlll}
\text{(Class)} & \mathtt{L} & ::= & \mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}}:\overline{\mathtt{V}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{N}\ \{\overline{\mathtt{M}}\} \\
\text{(Method)} & \mathtt{M} & ::= & \mathtt{def}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{V}})\{\mathtt{c}\}\mathtt{:V=e;} \\
\text{(Exp.)} & \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{this} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \mid \mathtt{e}\ \mathtt{as}\ \mathtt{T} \\
\text{(Par Type)} & \mathtt{U,V} & ::= & \mathtt{T}
\end{array}
\qquad
\begin{array}{rlll}
\text{(Type)} & \mathtt{S,T} & ::= & \mathtt{N} \mid \mathtt{T}\{\mathtt{c}\} \mid (\mathtt{x{:}S;T}) \\
\text{(N Type)} & \mathtt{N} & ::= & \mathtt{C} \mid \mathtt{N}\{\mathtt{c}\} \\
\text{(C Term)} & \mathtt{t} & ::= & \mathtt{x} \mid \mathtt{self} \mid \mathtt{this} \mid \mathtt{t.f} \mid \mathtt{new}\ \mathtt{C}(\bar{t}) \\
\text{(Const.)} & \mathtt{c,d} & ::= & \mathtt{true} \mid \mathtt{t{==}t} \mid \mathtt{c,c} \mid \mathtt{x{:}V;c}
\end{array}
$$

**FX well-formedness rules:**

$$
\Gamma \vdash \mathtt{true} : \mathtt{o} \quad \text{(TRUE)}
\qquad
\frac{\Gamma \vdash \mathtt{c}_0 : \mathtt{o} \qquad \Gamma \vdash \mathtt{c}_1 : \mathtt{o}}{\Gamma \vdash (\mathtt{c}_0, \mathtt{c}_1) : \mathtt{o}} \quad \text{(AND)}
\qquad
\frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \Gamma \vdash \mathtt{c}[\mathtt{t}/\mathtt{x}] : \mathtt{o}}{\Gamma \vdash \mathtt{x} : \mathtt{T}; \mathtt{c} : \mathtt{o}} \quad \text{(EXISTS)}
$$

$$
\frac{\Gamma \vdash \mathtt{class}(\mathtt{C})}{\Gamma \vdash \mathtt{C}\ \mathtt{type}} \quad \text{(CLASS)}
\qquad
\frac{\Gamma \vdash \mathtt{S}\ \mathtt{type}, \mathtt{T}\ \mathtt{type}}{\Gamma \vdash \mathtt{x} : \mathtt{S}; \mathtt{T}\ \mathtt{type}} \quad \text{(EXIST-T)}
\qquad
\frac{\Gamma \vdash \mathtt{T}\ \mathtt{type} \qquad \Gamma, \mathtt{self} : \mathtt{T} \vdash \mathtt{c} : \mathtt{o}}{\Gamma \vdash \mathtt{T}\{\mathtt{c}\}\ \mathtt{type}} \quad \text{(DEP)}
$$

**Type judgment rules:**

$$
\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}\{\mathtt{self} == \mathtt{x}\} \ \text{(T-VAR)}
\qquad
\frac{\Gamma \vdash \mathtt{e} : \mathtt{U} \qquad \Gamma \vdash \mathtt{T}\ \mathtt{type}}{\Gamma \vdash \mathtt{e}\ \mathtt{as}\ \mathtt{T} : \mathtt{T}} \ \text{(T-CAST)}
\qquad
\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \Gamma, \mathtt{z} : \mathtt{S} \vdash \mathtt{z}\ \mathtt{has}\ \mathtt{f} : \mathtt{V} \qquad (\mathtt{z}\ \text{fresh})}{\Gamma \vdash \mathtt{e.f} : (\mathtt{z} : \mathtt{S}; \mathtt{V}\{\mathtt{self} = \mathtt{z.f}\})} \ \text{(T-FIELD)}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{T}, \overline{\mathtt{e}} : \overline{\mathtt{V}} \\ \Gamma, \mathtt{v} : \mathtt{T}, \overline{\mathtt{v}} : \overline{\mathtt{V}} \vdash \mathtt{v}\ \mathtt{has}\ (\mathtt{m}(\overline{\mathtt{v}} : \overline{\mathtt{U}}), \mathtt{c} \to \mathtt{S}), \overline{\mathtt{V}} <: \overline{\mathtt{U}}, \mathtt{c} \qquad (\mathtt{v}, \overline{\mathtt{v}}\ \text{fresh})\end{array}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : (\mathtt{v} : \mathtt{T}; \overline{\mathtt{v}} : \overline{\mathtt{V}}; S)} \ \text{(T-INVK)}
\qquad
\frac{\begin{array}{c}\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \vdash \mathtt{class}(\mathtt{C}) \\ \Gamma, \mathtt{v} : \mathtt{C} \vdash \mathtt{fields}(\mathtt{v}) = \overline{\mathtt{f}} : \overline{\mathtt{V}} \qquad (\mathtt{v}, \overline{\mathtt{v}}\ \text{fresh}) \\ \Gamma, \mathtt{v} : \mathtt{C}, \overline{\mathtt{v}} : \overline{\mathtt{T}}, \mathtt{v}.\overline{\mathtt{f}} = \overline{\mathtt{v}} \vdash \overline{\mathtt{T}} <: \overline{\mathtt{V}}, \mathit{inv}(\mathtt{C}, \mathtt{v})\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}\{\overline{\mathtt{v}} : \overline{\mathtt{T}}; \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}) = \mathtt{self}, \mathit{inv}(\mathtt{C}, \mathtt{self})\}} \ \text{(T-NEW)}
$$

$$
\frac{\mathtt{this} : \mathtt{C} \vdash \mathtt{c} : o \qquad \mathtt{this} : \mathtt{C}, \overline{\mathtt{x}} : \overline{\mathtt{V}}, \mathtt{c} \vdash \mathtt{T}\ \mathtt{type}, \overline{\mathtt{V}}\ \mathtt{type}, \mathtt{e} : \mathtt{S}, \mathtt{S} <: \mathtt{T}}{\mathtt{def}\ \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{V}})\{\mathtt{c}\} : \mathtt{T} = \mathtt{e};\ \text{OK in } C} \ \text{(METHOD OK)}
\qquad
\frac{\overline{\mathtt{M}}\ \text{OK in C} \qquad \mathtt{this} : \mathtt{C} \vdash \mathtt{c} : o \qquad \mathtt{this} : \mathtt{C}, \mathtt{c} \vdash \overline{\mathtt{V}}\ \mathtt{type}, \mathtt{N}\ \mathtt{type}}{\mathtt{class}\ \mathtt{C}(\overline{\mathtt{f}} : \overline{\mathtt{V}})\{\mathtt{c}\}\ \mathtt{extends}\ \mathtt{N}\{\overline{\mathtt{M}}\}\ \text{OK}} \ \text{(CLASS OK)}
$$

**Transition rules:**

$$
\frac{\mathtt{x} : \mathtt{C} \vdash \mathtt{fields}(\mathtt{x}) = \overline{\mathtt{f}} : \overline{\mathtt{V}}}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{f}_\mathtt{i} \to \mathtt{e}_\mathtt{i}} \ \text{(R-FIELD)}
\qquad
\frac{\mathtt{x} : \mathtt{C} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{T} = \mathtt{e}}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}}) \to \mathtt{e}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}), \overline{\mathtt{d}}/\mathtt{this}, \overline{\mathtt{x}}]} \ \text{(R-INVK)}
$$

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

$$
\frac{\vdash \mathtt{C}\{\mathtt{self} == \mathtt{new}\ C(\overline{\mathtt{d}})\} <: \mathtt{T}}{(\mathtt{T})(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{d}})) \to \mathtt{new}\ C(\overline{\mathtt{d}})} \ \text{(R-CAST)}
\qquad
\frac{\mathtt{e}_i \to \mathtt{e}_i{}'}{\mathtt{e.m}(\ldots, \mathtt{e}_i, \ldots) \to \mathtt{e.m}(\ldots, \mathtt{e}'_i, \ldots)} \ \text{(RC-INVK-ARG)}
$$

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

**Figure 4.** Semantics of FX

"name" `x` (i.e. `e` is known statically to be equal to `x`, i.e. `e` is of type `T{self==x}`), then T-VAR lets us assert that `z:T{self==x}`, i.e. `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 casts provided inFJ in favor of a single cast, 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 Rule R-CAST).

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 S. Now we do not know the run-time value of `e`, so we shall assume that it is some fixed but unknown "proxy" value `z` (of type S) 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 `z` has a field `f` of type `V`[9], then we can assert that `e.f` has type `V` and, further, that it equals `z.f`. Hence, we can assert that `e.f` has type `(z:S;V{self=z.f})`.

[9] Note from the definition of `fields` in $O$ (Figure 3) that all occurrences of `this` in the declared type of the field `f` will have been replaced by `z`.

T-INVK has a very 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 `new C(`$\overline{\mathtt{v}}$`)` for some values $\overline{\mathtt{v}}$ of the given type.

***Operational semantics.*** The operational semantics is straightforward and essentially identical to FJ[25]. 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. 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 richer languages with richer notions of type this operation may involve run-time constraint solving using the fields of the object.

### 4.4 FX($\mathcal{G}$)

We now turn to showing how FGJ style generics can be supported in the FX family.

$\mathsf{FX}(\mathcal{G})$ is the language obtained by adding the following productions to $\mathsf{FX}(\cdot)$.

$$\begin{array}{rlll}
\text{(Par Type)} & \mathtt{V} & ::= & \mathtt{type} \\
\text{(Path)} & \mathtt{p} & ::= & \mathtt{x} \mid \mathtt{self} \mid \mathtt{this} \mid \mathtt{p.f} \\
\text{(Type)} & \mathtt{T} & ::= & \mathtt{X} \mid \mathtt{p} \\
\text{(C Term)} & \mathtt{t} & ::= & \mathtt{T} \\
\text{(Const.)} & \mathtt{c} & ::= & \mathtt{t} \trianglelefteq \mathtt{N} \mid \mathtt{t==t}
\end{array}$$

The constraint system specified by these changes will be called G.

That is, first we introduce the "type" `type`. FGJ method type parameters are modeled in $\mathsf{FX}(\mathcal{G})$ as normal parameters of type `type`.[10] 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 5.2. In brief, it permits powerful idioms using fixed but unknown types without requiring "wildcards".

Once fields of type `type` are permitted, it is natural to have *path* types (cf [43]). Such types name type-valued members of objects.

The set of types is now enhanced to permit some fixed but unknown types X. The key idea is that information about such types can be accumulated through constraints over $O$. Specifically we permit the constraint $\mathtt{t} \trianglelefteq \mathtt{N}$. It may be used, for instance, to specify upper bounds on type variables or fields (path types).[11]

The well-formedness and $\vdash_O$ rules in Figure 5 must be added.

EXAMPLE 4.1. *The* FGJ *parametric method:*

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

*can be represented as*

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

*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);
}
```

### 4.5 $\mathsf{FX}(\mathcal{A})$

We assume given a constraint system $\mathcal{A}$, with a vocabulary of predicates q and functions f. These are used to augment the constraints expressible in the language.

$$\begin{array}{rlll}
\text{(Type)} & \mathtt{T} & ::= & \text{new base types, e.g., } \mathtt{int}, \mathtt{boolean} \\
\text{(C Term)} & \mathtt{t} & ::= & \mathtt{f}(\overline{\mathtt{t}}) \\
\text{(Const.)} & \mathtt{c} & ::= & \mathtt{q}(\overline{\mathtt{t}})
\end{array}$$

The obvious rules are needed to ensure that formulas are well-formed.

[10] 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.

[11] To support structural typing, we permit the programmer to use `x has I` constraints, see Section 5.3.

$$\frac{\mathtt{p}(\overline{\mathtt{T}}):\mathtt{o} \in \mathcal{C} \qquad \Gamma \vdash \overline{\mathtt{t}}:\overline{\mathtt{T}}}{\Gamma \vdash \mathtt{p}(\overline{\mathtt{T}}):\mathtt{o}} \text{(PRED)} \qquad \frac{\mathtt{f}(\overline{\mathtt{T}}):\mathtt{T} \in \mathcal{C} \qquad \Gamma \vdash \overline{\mathtt{t}}:\overline{\mathtt{T}}}{\Gamma \vdash \mathtt{f}(\overline{\mathtt{T}}):\mathtt{T}} \text{(FUN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_0 \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \\ (\Gamma \vdash \mathtt{T}_0 <: \mathtt{T}_1 \vee \Gamma \vdash \mathtt{T}_1 <: \mathtt{T}_0)\end{array}}{\Gamma \vdash \mathtt{t}_0 = \mathtt{t}_1 : \mathtt{o}} \text{(EQUALS)}$$

No additional type judgments or transition rules are needed.

### 4.6 $\mathsf{FX}(\mathcal{G},\mathcal{A})$

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.

### 4.7 Results

The following results hold for $\mathsf{FX}(\mathcal{G},\mathcal{A})$.

THEOREM 4.1 (Subject Reduction). *If* $\Gamma \vdash \mathtt{e}:\mathtt{T}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$ *then for some type* $\mathtt{S}$, $\Gamma \vdash \mathtt{e}' : \mathtt{S}, S <: T$.

The theorem needs the Substitution Lemma:

LEMMA 4.2. *The following is a derived rule:*

$$\frac{\Gamma \vdash \overline{\mathtt{d}}:\overline{\mathtt{U}} \qquad \Gamma,\overline{\mathtt{x}}:\overline{\mathtt{U}} \vdash \overline{\mathtt{U}} <: \overline{\mathtt{V}} \qquad \Gamma,\overline{\mathtt{x}}:\overline{\mathtt{V}} \vdash \mathtt{e}:\mathtt{T}}{\Gamma \vdash \mathtt{e}[\overline{\mathtt{d}}/\overline{\mathtt{x}}] : \mathtt{S}, \mathtt{S} <: \overline{\mathtt{x}}:\overline{\mathtt{A}};\mathtt{T}} \text{(SUBST)}$$

We let values be of the form $\mathtt{v} ::= \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$.

THEOREM 4.3 (Progress). *If* $\vdash \mathtt{e}:\mathtt{T}$ *then one of the following conditions holds:*

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

THEOREM 4.4 (Type soundness). *If* $\vdash \mathtt{e}:\mathtt{T}$ *and* $\mathtt{e}$ *reduces to a normal form* $\mathtt{e}'$ *then either* $\mathtt{e}'$ *is a value* $\mathtt{v}$ *and* $\vdash \mathtt{v}:\mathtt{S},\mathtt{S} <: \mathtt{T}$ *or* $\mathtt{e}'$ *contains a stuck cast sub-expression.*

## 5. Discussion and related work

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 classes* [34, 35, 17]. are a language-based extensibility mechanism that where originally introduced in the language BETA [34] as a mechanism for supporting genericity. Virtual classes in BETA are not statically type safe, but this has been remedied in recent formulations [16, 17] and in variants of virtual classes [43, 42, 11, 27] using path-dependent types.

*Virtual types*, also introduced in BETA [34], are similar to virtual classes. A virtual type is a type binding nested within an enclosing instance. Virtual types may be used to provide genericity; indeed Thorup [57] proposed extending Java with virtual types as a genericity mechanism. Virtual types influenced Java's wildcards [59, 22, 8].

Type properties share many similarities with virtual types [35, 34, 16, 17, 11] and similar constructs built on path-dependent types found in languages such as Scala [43], and J& [42]. Indeed, one of the first proposals for adding genericity to Java was via

$$\frac{\Gamma \vdash \mathtt{p : T} \qquad \Gamma, \mathtt{x : T} \vdash \mathtt{x\ has\ X:\ type}}{\Gamma \vdash \mathtt{p.X\ type}} \ (\textsc{Path}) \qquad \Gamma, \mathtt{X : type} \vdash \mathtt{X\ type} \quad (\textsc{Type-Var}) \qquad \mathtt{S == T} \vdash_{\mathcal{O}} S \trianglelefteq T, T \trianglelefteq S \quad (\textsc{Equals})$$

$$\frac{\Gamma \vdash_{\mathcal{O}} \mathtt{p} \trianglelefteq \mathtt{T} \qquad \Gamma, \mathtt{x : T} \vdash_{\mathcal{O}} \mathtt{x\ has\ I}}{\Gamma, \mathtt{x : p} \vdash_{\mathcal{O}} \mathtt{x\ has\ I}} \ (\textsc{Inh-P}) \qquad \frac{\Gamma \vdash_{\mathcal{O}} \mathtt{X} \trianglelefteq \mathtt{T} \qquad \Gamma, \mathtt{x : T} \vdash_{\mathcal{O}} \mathtt{x\ has\ I}}{\Gamma, \mathtt{x : X} \vdash_{\mathcal{O}} \mathtt{x\ has\ I}} \ (\textsc{Inh-X})$$

**Figure 5.** $\mathsf{FX}(\mathcal{G})$ semantics

virtual types [57], and Java wildcards (i.e., parameters with use-site variance) were developed from a line of work beginning with virtual types [58, 26, 59].

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 [57] 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) { ... }
}
```

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

```
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 = { ... }
}
```

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 Type parameters and wildcards

Type properties are also similar, but not identical to, type parameters. The key difference is that type properties are instance members bound during object construction. Type properties are thus accessible through expressions: `e.T` is a legal type.

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

Wildcards in Java [22, 59] can be motivated by the following example. 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<?>`; that is the type of all `Set` instantiated on *some* parameter. Clients of this type do not know what parameter the actual instance of `Set` is bound to, which restricts the methods that can be invoked on the object.

Wildcards can also be bounded above and below with `? extends T` and `? super T` respectively. In X10, a similar effect is achieved by leaving the element type property of `Set` partially unconstrained.

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:

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

Through such a translation, the $\mathsf{FX}(\mathcal{G})$ calculus in Section 4 captures the essence of Java's wildcards, but extended with support for run-time type introspection.

Constrained types also support *proper abstraction* [59]. 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

Type constraints need not be limited to subtyping constraints. By introducing structural constraints on types, one can instantiate type properties on any type with a given set of methods and 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 in separate libraries since it does not require code of one library to implement an interface to satisfy a constraint of another library.

We consider an extension of the X10 type system to support structural type constraints. Formally, the extension is straightforward; indeed the FX family already supports structural constraints via the rules for "`x has I`". Figure 3 and Figure 5.

Structural constraints on types are found in many languages. Haskell supports type classes [30, 23]. In Modula-3, type equivalence and subtyping are structural rather than nominal as in object-oriented languages of the C family such as C++, Java, Scala, and X10. The language PolyJ [40] allows type parameters to be bounded using structural *where clauses* [14]. For example, a sorted list class could be 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 def 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. Consider 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 two Ts and returns an int.

Using this definition, one can create lists with distinct types of, for example, case-sensitive and case-insensitive strings:

```
val unixFiles
  = new SortedList[String]
        (String.compareTo.(String));
val windowsFiles
  = new SortedList[String]
        (String.compareToIgnoreCase.(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 files into a function that expects a list of Windows files, and vice versa.

#### 5.3.2 Optional methods

Structural method constraints permit the introduction of CLU-style optional methods [33]. Consider the following Array class:

```
class Array[T] {
  def add(a: Array[S])
    {T has add(S): U}: Array[U] = { ... }
  ...
}
```

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 Related work

Constraint-based type systems, dependent types, and generic types have been well-studied in the literature.

***Constraint-based type systems.*** The use of constraints for type inference and subtyping has a history going back to Mitchell [39] and by Reynolds [48]. These and subsequent systems are based on constraints over types, but not over values. Trifonov and Smith [60] proposed a type system in which types are refined using subtyping constraints. Pottier [47] presents a constraint-based type system for an ML-like language with subtyping. These developments lead to HM($X$) [54], 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.

Sulzmann and Stuckey [55] showed that the type inference algorithm for HM($X$) can be encoded as a constraint logic program parametrized by the constraint system $X$. This is very much in spirit with our approach. Constrained types open the door to *user-defined* predicates and functions, effectively permitting the user to enrich $\mathcal{C}$ (hence the power of the compile-time type-checker) with application-specific constraints using a constraint programming language such as CLP($\mathcal{C}$) [28] or the richer RCC($\mathcal{C}$) [29].

***Dependent types.*** Dependent type systems [63, 13, 37, 6] parametrize types on values. Refinement type systems [21, 1, 31, 24, 18, 19, 49], introduced by Freeman and Pfenning [21], are dependent type systems that extend a base type system through constraints on values. These systems do not treat value and type constraints uniformly.

Our work is closely related to DML, [63], an extension of ML with dependent types. DML is also built parametrically on a constraint solver. Types are refinement types; 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).

The most obvious distinction between DML and constrained types lies in the target domain: DML is designed for functional programming whereas constrained types are designed for imperative, concurrent object-oriented languages. But there are several other crucial differences as well.

DML achieves its separation between compile-time and run-time processing by not permitting program variables to be used in types. Instead, a parallel set of (universally or existentially quantified) "index" variables are introduced. Second, DML permits only variables of basic index sorts known to the constraint solver (e.g., bool, int, nat) to occur in types. In contrast, constrained types permit program variables at any type to occur in constrained types. As with DML only operations specified by the constraint system are permitted in types. However, these operations always include field selection and equality on object references. Note that DML-style constraints are easily encoded in constrained types.

Logically qualified types, or liquid types [49], 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 and if one type's qualifier implies the other's. The Hindley–Milner type inference algorithm is used to infer base types; these types are used as templates for inference of the liquid types. The types of certain expressions are over-approximated to ensure inference is decidable. To improve precision of the inference algorithm, and hence to reduce the annotation burden on the programmer, the type system is path sensitive.

Hybrid type-checking [18, 19] introduced another refinement type system. While typing is undecidable, dynamic checks are inserted into the program when necessary if the type-checker (which includes a constraint solver) cannot determine type safety statically. In FX($\mathcal{G}$), dynamic type checks, including tests of dependent constraints, are inserted only at explicit casts or instanceof expressions; constraint solving is performed at compile time.

Concoqtion [20] extends types in OCaml [32] with constraints written as Coq [12] 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. Concoqtion encodes OCaml types and value to allow reasoning in the Coq formulae; however, there is an impedance mismatch caused by the differing syntax, representation, and behavior of OCaml versus Coq.

***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 [7, 45, 40, 57, 2]. GJ [7] implements invariant type parameters via type erasure. PolyJ [40] 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 [62, 61] also support a run-time representation of types, using Java's reflection API. NextGen [9, 2] 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♯ also supports generic types via run-time instantiation in the CLR [56]. Type parameters may be declared with definition-site variance tags. Generalized type constraints were proposed for C♯ [15]. 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.

## 6. Conclusions

We have presented a constraint-based framework FX for type- and value-dependent types in an object-oriented language. The framework captures the essence of object-oriented type systems in a constraint system $O$. Instantiations of the framework augment the core object-oriented features with generic types and dependent types. 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 $\mathsf{FX}(\mathcal{G}, \mathcal{A})$ 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.

## Acknowledgments

The authors thank Bob Blainey, Doug Lea, Jens Palsberg, Lex Spoon, and Olivier Tardieu for valuable feedback on versions of the language. We thank Andrew Myers and Michael Clarkson for providing us with their implementation of PolyJ, on which our implementation was based, and for many discussions over the years about parametrized types in Java.

## References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 163–173, January 1994.

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

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

[4] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. `http://www.e-pig.org/downloads/ydtm.pdf`, April 2005.

[5] David Aspinall and Martin Hofmann. *Dependent Types*, chapter 2. In Pierce [46], 2004.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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, New York, NY, USA, 2005. ACM.

[11] 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, New York, NY, USA, 2007. ACM Press.

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

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

[14] 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).

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

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

[17] 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.

[18] Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd Annual Symposium on Principles of Programming Languages (POPL'06)*, pages 245–256, 2006.

[19] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 2006.

[20] 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.

[21] 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.

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

[23] 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.

[24] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 410–423, 1996.

[25] 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.

[26] 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.

[27] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 113–132, New York, NY, USA, 2007. ACM.

[28] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119. ACM Press, New York (NY), USA, 1987.

[29] Radha Jagadeesan, Gopalan Nadathur, and Vijay A. Saraswat. Testing concurrent systems: An interpretation of intuitionistic logic. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS*, volume 3821 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 2005.

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

[31] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

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

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

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

[35] 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.

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

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

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

[39] 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.

[40] 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.

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

[42] 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 '06)*, pages 21–36, Portland, OR, October 2006.

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

[44] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 201–224. Springer-Verlag, July 2003.

[45] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 146–159, Paris, France, January 1997.

[46] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.

[47] François Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, 1996.

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

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

[50] 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.

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

[52] Vijay Saraswat et al. The X10 language specification. Technical report, IBM T.J. Watson Research Center, 2006.

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

[54] 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.

[55] Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming*, 18(2):251–283, 2008.

[56] 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.

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

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

[59] 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.

[60] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Third International Static Analysis Symposium (SAS)*, number 1145 in LNCS, pages 349–365, 1996.

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

[62] 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.

[63] 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.