

# 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 for 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  $C$ . 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  $C$  on the immutable instance state of  $C$  (the *properties*). Many (value-)dependent type systems for OO languages can be cast 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  $FXC$  of programming languages with a common set of sound type-checking rules. By varying  $C$ , 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, 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 [58, ?, 57] 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*. A place encapsulates data and one or more *activities* that operate on the data. Places capture the idea of *locality* (data in the same place is “close”, data in a different place is “far”) and *heterogeneity* (one place may target for a collection of

tightly integrated cores, another may target a GPGPU). Data in all places 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 (tasks) 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* and may support an arbitrary base type, and accesses through *points* which 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 arrays over rectangular regions, it is important that this information be conveyed statically (through the type system) so that efficient code can be generated. 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 read and write accesses by multiple activities to different divisions.

These requirements motivated us to develop a framework for dependent types in OO languages [55]. *Dependent type systems* [38, 68, 46, 5, 6, 4, 15] 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 [56] 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, the principal insight is that objects typically have some immutable state constraints on which 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,  $\text{Array}\{\text{self.length}==N\}$  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  $D\{d\}$  (in the sense that any value satisfying  $C\{c\}$  may be used wherever a value satisfying  $D\{d\}$  is expected) 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

Constrained types maintain a key phase distinction between compile-time computation (entailment checking in the underlying constraint system) and run-time (normal OO computation). Dynamic type casting is permitted—it leads to code being generated for execution at run time to check that 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)[68]. Constrained types are a very 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). Finally, constrained types open the door to *user-defined* predicates and functions, effectively permitting the user to enrich  $C$  (hence the power of the compile-time type-checker) by developing application-specific constraints using a constraint programming language such as CLP( $C$ ) [31] or the richer RCC( $C$ ) [?].

## 1.1 Generic types

In this paper we extend the constrained types approach to handle *generic types* [35, 43, 7, 42, 62, 24, 61]—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*.<sup>1</sup> 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) members. Members are of two kinds. A *field* has a name and an associated type. A *method* has a name and an associated signature (the type of arguments and the return type). The members available at a type and its supertype are related: members “flow” from supertypes to subtypes in a manner determined by the over-riding rules of the language.

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  $T \text{ has } f:T$  and  $T \text{ has } m(\bar{x}:S):T$  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  $\sqsubseteq$  relation must be transitive. If the user program does not specify that a class  $C$  extends  $D$  (transitively), then  $C$  does *not* extend  $D$  (closed world assumption). Section 3.1 describes the resulting Object constraint system,  $O$ , in more detail.

The next question is: where should variable types be permitted? Clearly, one must permit methods to have type parameters (as permitted for instance in Java [24], Scala [45] and in functional languages such as ML [40] and Haskell [32]). 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* (cf [24]), 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 run time—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.

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.<sup>2</sup>

The type `Array{self.T==int}` specifies the type of all arrays whose base type ( $T$ ) is `int`. The type `Array{self.T<: 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 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 (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 the `Array.get`).

Many features of modern object-oriented type systems fall out naturally from this type system.

## 1.2 The FX( $C$ ) family

To further the foundations of constrained type, we develop the FX family of languages. The core expression language is essentially FJ[27], 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.  $FX()$  is  $FX$  instantiated over the vacuous constraint system—the only user-specifiable constraint permitted is the vacuous `true`.  $FX()$  corresponds to FJ.  $FX(G)$  permits user-specifiable constraints from the Object constraint system  $O$  described above—it corresponds to FGJ (being somewhat richer in permitting path types).  $FX(D(\mathcal{A}))$  permits the use of constraints from some constraint system  $\mathcal{A}$ —it corresponds to a language with a pure value-dependent type system as described in [?].  $FX(G, D(\mathcal{A}))$  permits the use of constraints from  $O$  and  $\mathcal{A}$ —thus permitting value- and type-dependence—and is the main topic for this paper.

<sup>2</sup> 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.

<sup>1</sup> This can be extended naturally to account for interfaces.

### 1.3 Design and implementation of X10 type system

$FX(G, D(\mathcal{A}))$  is the basis for the design of the type system for X10 v1.7, a new version of the language scheduled to be released in Fall 08. We discuss the details of the design. We describe the implementation. 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.

**Contributions.** We extend the constrained types approach to handle generic types. We present a family of formal languages,  $FX(C)$  that capture the essence of the idea of constrained types. By appropriately choosing  $C$ , 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. 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.

**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. We discuss extensions of the type system, including for virtual types and self types, in Section 4. The implementation of generics in X10 by translation to Java is described in Section 5. Section 6 discusses related work. Finally, Section 7 concludes.

## 2. X10 language overview

This section presents an informal description of dependent and generic types in X10. The type system is formalized in a simplified version of X10,  $FX$ , in Section 3.

When presenting syntax, we follow the usual conventions for Featherweight Java: we write  $\bar{t}$  for the list  $t_1, \dots, t_n$ ; terms with list subterms are considered a single list of terms (e.g., we write  $\bar{x}.\bar{T}$  for the list  $x_1:T_1, \dots, x_n:T_n$ ).

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.

The compiler checks that constraints are expressions of type boolean and that they can be statically checked by the compiler's constraint solver. X10 supports conjunctions of equalities over final variables and compile-time constants, and equalities and subtyping constraints over types. Compiler plugins may be installed to handle richer constraint systems such as Presburger arithmetic or set constraints.

For brevity, the constraint may be omitted and interpreted as true. The syntax  $C[\bar{T}](\bar{e})$  is shorthand for  $C[\bar{X}=\bar{T}](\bar{x}=\bar{e})$  where  $X_i$  are the type properties and  $x_i$  are the value properties of  $C$ . If either list of properties is empty, it may be omitted.

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

```
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

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 *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.

```

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

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 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 5.

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`).<sup>3</sup>

<sup>3</sup> Actual type arguments can be inferred from the types of the value arguments. Type inference is out of the scope of this paper.

```

class SortedList(compare: (T,T)=>int) extends List {
  def this[T](hd: T, tl: List[T],
    compare: (T,T)=>int)
    : SortedList[T](compare) = {
    super[T](hd, tl);
    property(compare);
  }

  def this[T](hd: T, tl: List[T]){T <: Comparable}
    : SortedList[T](T.compare.(Object)) = {
    this[T](hd, tl, T.compareTo.(Object));
  }

  def add(x: T) = {
    ... compare(x, y) ...
  }
}

```

**Figure 3.** A `SortedList` class with function-typed value properties

`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

Type properties and subtyping constraints may be used in X10 to provide use-site variance constraints [29].

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`.
- `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}`.

### 2.4 Function-typed properties

X10 supports first-class functions. Function-typed properties are a useful feature for generic collection classes. Consider the definition of the `SortedList` class in Figure 3. The class has a property `compare` of type `(T,T)=>int`—a function that takes two `T`s and

returns an `int`. The class declares two constructors, one that takes a function to bind to the `compare` property, and another that binds `T`'s `compare` method to the property. The `compare` method uses the `equals` function to compare elements.

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.

### 3. Semantics

We now describe the semantics of languages in the FX family.

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  $\mathcal{X}$ .

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

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

Given  $\mathcal{L}$ , its input constraint system  $\mathcal{X}$  we now show how to define  $\mathcal{O}$ . The inference relation for  $\mathcal{O}$  depends on the object-oriented structure of the input program  $P$  in  $\mathcal{L}$ . 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 constraints of  $\mathcal{O}$  are given by:

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

$\text{class}(C)$  is intended to be true for all classes  $C$  defined in the program.  $S \sqsubseteq T$  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  $x$ ,  $\text{fields}(x)$  is intended to specify the (complete) set of typed fields available to  $x$ .  $x \text{ has } I$  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 following axioms and inference rules in Figure 4. Since  $\mathcal{O}$  is a constraint system [?], 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  $C \text{ has } I$ , for classes  $C, D$  and members  $I$ ; that is, if the rules cannot be used to establish  $\vdash_{\mathcal{O}} C <: D$  ( $\vdash_{\mathcal{O}} C \text{ has } I$ ), then it is the case that  $\vdash_{\mathcal{O}} C \not<: D$  ( $\vdash_{\mathcal{O}} \neg(C \text{ has } I)$ ).

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).

#### 3.2 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:  $\Gamma \vdash T$  **type**, (2) Subtyping:  $\Gamma \vdash S <: T$ , (3) Typing:  $\Gamma \vdash e : T$ , (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:  $\Gamma \vdash C1$  OK (class definition  $C1$  is well-formed).

In defining these judgments we will use  $\Gamma \vdash_C c$ , the judgment corresponding to the underlying constraint system. For simplicity, we define  $\Gamma \vdash c$  to mean  $\sigma(\Gamma) \vdash_C c$ , where the *constraint projection*,  $\sigma(\Gamma)$  is defined as follows.

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

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).

#### 3.3 FX()

The semantics of  $\text{FX}()$  is presented in Figure 5.

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 class, 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.<sup>4</sup>

The syntax for constraints in  $\text{FX}()$  is specified in Figure 5. We distinguish a subset of these constraints as *user constraints*—these are permitted to occur in programs. For  $\text{FX}()$  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 5 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.

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 sub-

<sup>4</sup> 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.

$$\begin{array}{c}
\frac{\text{class } C(\bar{f} : \bar{V}) \text{ extends } D \dots \in P}{\vdash_O \text{class}(C), C \leq D} \text{ (CLASS)} \quad \frac{\Gamma \vdash_O \text{fields}(x) = \bar{f} : \bar{V}}{\Gamma \vdash_O x \text{ has } f_1 : V_1} \text{ (HAS-F)} \quad \vdash_O \text{new } D(\bar{v}).f_i = t_i \text{ (SEL)} \quad \frac{\Gamma \vdash_O x : C, \text{class}(C)}{\Gamma \vdash_O \text{inv}(C, x)} \text{ (INV)} \\
\\
\vdash_O T \leq T \text{ (V-ID)} \quad \frac{\Gamma \vdash_O t \leq T}{\Gamma \vdash_O t < : T} \text{ (SUB-X)} \quad \frac{\Gamma \vdash_O T_1 < : T_2, T_2 < : T_3}{\Gamma \vdash_O T_1 < : T_3} \text{ (S-TRANS)} \\
\\
\frac{\Gamma, c \vdash_O S < : T}{\Gamma \vdash_O S\{c\} < : T} \text{ (S-CONST-L)} \quad \frac{\Gamma \vdash_O S < : T \quad \Gamma, \text{self} : S \vdash_O c}{\Gamma \vdash_O S < : T\{c\}} \text{ (S-CONST-R)} \quad \frac{\text{class } C(\dots) \text{ extends } D\{\dots\} \in P}{\vdash_O C < : D} \text{ (S-EXTENDS)} \\
\\
\frac{\Gamma \vdash U \text{ type} \quad \Gamma \vdash_O S < : T \quad (x \text{ fresh})}{\Gamma \vdash_O x : U; S < : T} \text{ (S-EXISTS-L)} \quad \frac{\Gamma \vdash t : U \quad \Gamma \vdash_O S < : T[t/x]}{\Gamma \vdash_O S < : x; U : T} \text{ (S-EXISTS-R)} \quad \frac{\Gamma \vdash_O S < : T \quad \Gamma \vdash_O T < : S}{\Gamma \vdash_O S \equiv T} \text{ (TYPE-EQUIV)} \\
\\
x : \text{Object} \vdash_O \text{fields}(x) = \bullet \text{ (FIELDS-B)} \quad \frac{\Gamma, x : D \vdash_O \text{fields}(x) = \bar{g} : \bar{V} \quad \text{class } C(\bar{f} : \bar{U})\{c\} \text{ extends } D\{\bar{M}\} \in C}{\Gamma, x : C \vdash \text{fields}(x) = \bar{g} : \bar{V}, \bar{f} : U[x/\text{this}]} \text{ (FIELDS-I)} \quad \frac{\Gamma, x : S \vdash_O \text{fields}(x) = \bar{f} : \bar{V}}{\Gamma, x : S\{d\} \vdash_O \text{fields}(x) = \bar{f} : \bar{V}[d[x/\text{self}]]} \quad \frac{\Gamma, x : (y : U; S) \vdash_O \text{fields}(x) = \bar{f} : \bar{V}}{\Gamma, x : (y : U; S) \vdash_O \text{fields}(x) = \bar{f} : \bar{V}} \text{ (FIELDS-C,E)} \\
\\
\frac{\Gamma, x : C \vdash_O \text{class}(C) \quad \theta = [x/\text{this}] \quad \text{def } m(\bar{z} : \bar{V})\{c\} : T = e \in P}{\Gamma, x : C \vdash_O x \text{ has } (m(\bar{z} : \bar{V})\{c\}) : T\theta = e} \text{ (METHOD-B)} \quad \frac{\Gamma, x : D \vdash_O x \text{ has } m(\bar{z} : \bar{V})\{c\} : T = e \quad \text{class } C(\dots) \text{ extends } D\{\bar{M}\} \quad m \notin \bar{M}}{\Gamma, x : C \vdash_O x \text{ has } m(\bar{z} : \bar{V})\{c\} : T = e} \text{ (METHOD-I)} \quad \frac{\Gamma, x : S \vdash_O x \text{ has } m(\bar{z} : \bar{V})\{c\} : T = e}{\Gamma, x : S\{d\} \vdash_O x \text{ has } m(\bar{z} : \bar{V})\{c\} : T\{d[x/\text{self}]\} = e} \quad \frac{\Gamma, x : (y : U; S) \vdash_O x \text{ has } m(\bar{z} : \bar{V})\{c\} : (y : U; T) = e}{\Gamma, x : (y : U; S) \vdash_O x \text{ has } m(\bar{z} : \bar{V})\{c\} : (y : U; T) = e} \text{ (METHOD-C,E)}
\end{array}$$

Figure 4. The Object constraint system,  $O$

type of  $C$ ; it is of type  $T\{c\}$  if it is of type  $T$  and it satisfies the constraint  $c[o/\text{self}]$ <sup>5</sup>; 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  $\text{self}==x$  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 “name”  $x$  (i.e.  $e$  is known statically to be equal to  $x$ , i.e.  $e$  is of type  $T\{\text{self}==x\}$ ), then T-VAR lets us assert that  $z : T\{\text{self}==x\}$ , 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 in FJ 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  $\Gamma$  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  $\Gamma$ ). If we can establish that  $z$

has a field  $f$  of type  $V$ <sup>6</sup>, 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  $(z : S; V\{\text{self}=z.f\})$ .

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  $\text{new } C(\bar{v})$  for some values  $\bar{v}$  of the given type.

**Operational semantics.** The operational semantics is straightforward and essentially identical to FJ[27]. 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  $\text{FX}()$ , 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.

### 3.4 $\text{FX}(G)$

We now turn to showing how FGJ style generics can be supported in the  $\text{FX}$  family.

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

<sup>5</sup> Thus the constraint  $c$  in a type  $T\{c\}$  should be thought of as a unary predicate  $\lambda \text{self}.c$ , an object is of this type if it is of type  $T$  and satisfies this predicate.

<sup>6</sup> Note from the definition of  $\text{fields}$  in  $O$  (Figure 4) that all occurrences of  $\text{this}$  in the declared type of the field  $f$  will have been replaced by  $z$ .

**FX productions:**

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

**FX well-formedness rules:**

$\frac{}{\Gamma \vdash \text{true} : o}$ (TRUE)	$\frac{\Gamma \vdash c_0 : o \quad \Gamma \vdash c_1 : o}{\Gamma \vdash (c_0, c_1) : o}$ (AND)	$\frac{\Gamma \vdash t : T \quad \Gamma \vdash c[t/x] : o}{\Gamma \vdash x : T; c : o}$ (EXISTS)
$\frac{\Gamma \vdash \text{class}(C)}{\Gamma \vdash C \text{ type}}$ (CLASS)	$\frac{\Gamma \vdash S \text{ type}, T \text{ type}}{\Gamma \vdash x : S; T \text{ type}}$ (EXIST-T)	$\frac{\Gamma \vdash T \text{ type} \quad \Gamma, \text{self} : T \vdash c : o}{\Gamma \vdash T\{c\} \text{ type}}$ (DEP)

**Type judgment rules:**

$\Gamma, x : T \vdash x : T\{\text{self} == x\}$ (T-VAR)	$\frac{\Gamma \vdash e : U \quad \Gamma \vdash T \text{ type}}{\Gamma \vdash e \text{ as } T : T}$ (T-CAST)	$\frac{\Gamma \vdash e : S \quad \Gamma, z : S \vdash z \text{ has } f : V \quad (z \text{ fresh})}{\Gamma \vdash e.f : (z : S; V\{\text{self} = z.f\})}$ (T-FIELD)
$\frac{\Gamma \vdash e : T, \bar{e} : \bar{V} \quad \Gamma, v : T, \bar{v} : \bar{V} \vdash v \text{ has } (m(\bar{v} : \bar{U}), c \rightarrow S), \bar{V} <: \bar{U}, c \quad (v, \bar{v} \text{ fresh})}{\Gamma \vdash e.m(\bar{e}) : (v : T; \bar{v} : \bar{V}; S)}$ (T-INVK)	$\frac{\Gamma \vdash \bar{e} : \bar{T} \quad \vdash \text{class}(C) \quad \Gamma, v : C \vdash \text{fields}(v) = \bar{f} : \bar{V} \quad (v, \bar{v} \text{ fresh}) \quad \Gamma, v : C, \bar{v} : \bar{T}, v.f = \bar{v} \vdash \bar{T} <: \bar{V}, \text{inv}(C, v)}{\Gamma \vdash \text{new } C(\bar{e}) : C\{\bar{v} : \bar{T}; \text{new } C(\bar{v}) = \text{self}, \text{inv}(C, \text{self})\}}$ (T-NEW)	
$\frac{\text{this} : C \vdash c : o \quad \text{this} : C, \bar{x} : \bar{V}, c \vdash T \text{ type}, \bar{V} \text{ type}, e : S, S <: T}{\text{def } m(\bar{x} : \bar{V})\{c\} : T = e; \text{ OK in } C}$ (METHOD OK)	$\frac{\bar{M} \text{ OK in } C \quad \text{this} : C \vdash c : o \quad \text{this} : C, c \vdash \bar{V} \text{ type}, N \text{ type}}{\text{class } C(\bar{f} : \bar{V})\{c\} \text{ extends } N\{\bar{M}\} \text{ OK}}$ (CLASS OK)	

**Transition rules:**

$\frac{x : C \vdash \text{fields}(x) = \bar{f} : \bar{V}}{(\text{new } C(\bar{e})).f_i \rightarrow e_i}$ (R-FIELD)	$\frac{x : C \vdash x \text{ has } m(\bar{x} : \bar{T})\{c\} : T = e}{(\text{new } C(\bar{e})).m(\bar{d}) \rightarrow e[\text{new } C(\bar{e}), \bar{d}/\text{this}, \bar{x}]}$ (R-INVK)
$\frac{e \rightarrow e'}{e.f_i \rightarrow e'.f_i}$ (RC-FIELD)	$\frac{e \rightarrow e'}{e.m(\bar{e}) \rightarrow e'.m(\bar{e})}$ (RC-INVK-RECV)
$\frac{\vdash C\{\text{self} == \text{new } C(\bar{d})\} <: T}{(T)(\text{new } C(\bar{d})) \rightarrow \text{new } C(\bar{d})}$ (R-CAST)	$\frac{e_i \rightarrow e'_i}{e.m(\dots, e_i, \dots) \rightarrow e.m(\dots, e'_i, \dots)}$ (RC-INVK-ARG)
$\frac{e \rightarrow e'}{(T)e \rightarrow (T)e'}$ (RC-CAST)	$\frac{e_i \rightarrow e'_i}{\text{new } C(\dots, e_i, \dots) \rightarrow \text{new } C(\dots, e'_i, \dots)}$ (RC-NEW-ARG)

**Figure 5.** Semantics of FX

(Par Type)  $V ::= \text{type}$   
 (Path)  $p ::= x \mid \text{self} \mid \text{this} \mid p.f$   
 (Type)  $T ::= X \mid p$   
 (C Term)  $t ::= T$   
 (Const.)  $c ::= t \leq N \mid t == t$

That is, first we introduce the “type” type. FGJ method type parameters are modeled in  $\text{FX}(G)$  as normal parameters of type `type`.<sup>7</sup> 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 4.2. 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 [45]). 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  $t \leq N$ . It may be used, for instance, to specify upper bounds on type variables or fields (path types).<sup>8</sup>

The well-formedness and  $\vdash_O$  rules in Figure 6 must be added.

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;
```

The 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);
}
```

### 3.5 $\text{FX}(D(\mathcal{A}))$

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.

(Type)  $T ::=$  new base types, e.g., `int`, `boolean`  
 (C Term)  $t ::= f(\bar{t})$   
 (Const.)  $c ::= q(\bar{t})$

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

$$\frac{p(\bar{T}) : o \in C \quad \Gamma \vdash \bar{t} : \bar{T}}{\Gamma \vdash p(\bar{T}) : o} \quad (\text{PRED})$$

$$\frac{f(\bar{T}) : T \in C \quad \Gamma \vdash \bar{t} : \bar{T}}{\Gamma \vdash f(\bar{T}) : T} \quad (\text{FUN})$$

<sup>7</sup>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.

<sup>8</sup>To support structural typing, we permit the programmer to use `x has I` constraints, see Section 4.4.

$$\frac{\Gamma \vdash t_0 : T_0 \quad \Gamma \vdash t_1 : T_1 \quad (\Gamma \vdash T_0 <: T_1 \vee \Gamma \vdash T_1 <: T_0)}{\Gamma \vdash t_0 = t_1 : o} \quad (\text{EQUALS})$$

No additional type judgments or transition rules are needed.

### 3.6 $\text{FX}(G, D(\mathcal{A}))$

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

### 3.7 Results

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

THEOREM 3.1 (Subject Reduction). *If  $\Gamma \vdash e : T$  and  $e \rightarrow e'$  then for some type  $S$ ,  $\Gamma \vdash e' : S, S <: T$ .*

The theorem needs the Substitution Lemma:

LEMMA 3.2. *The following is a derived rule:*

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

We let values be of the form  $v ::= \text{newC}(\bar{v})$ .

THEOREM 3.3 (Progress). *If  $\vdash e : T$  then one of the following conditions holds:*

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

THEOREM 3.4 (Type soundness). *If  $\vdash e : T$  and  $e$  reduces to a normal form  $e'$  then either  $e'$  is a value  $v$  and  $\vdash v : S, S <: T$  or  $e'$  contains a stuck cast sub-expression.*

## 4. Discussion

Generic constrained types generalize virtual types and have a connection to parametric types with use-site variance annotations, such as Java’s wildcards.

### 4.1 Virtual types

Type properties share many similarities with virtual types [37, 36, 18, 19, 12] and similar constructs built on path-dependent types found in languages such as Scala [45], and J& [44]. Indeed, one of the first proposals for adding genericity to Java was via virtual types [62], and Java wildcards (i.e., parameters with use-site variance) were developed from a line of work beginning with virtual types [63, 29, 64].

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

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

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

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



$$\begin{array}{c}
\frac{\Gamma \vdash p : T \quad \Gamma, x : T \vdash x \text{ has } X : \text{type}}{\Gamma \vdash p.X \text{ type}} \text{ (PATH)} \qquad \Gamma, X : \text{type} \vdash X \text{ type} \text{ (TYPE-VAR)} \qquad S == T \vdash_O S \leq T, T \leq S \text{ (EQUALS)} \\
\\
\frac{\Gamma \vdash_O p \leq T \quad \Gamma, x : T \vdash_O x \text{ has } I}{\Gamma, x : p \vdash_O x \text{ has } I} \text{ (INH-P)} \qquad \frac{\Gamma \vdash_O X \leq T \quad \Gamma, x : T \vdash_O x \text{ has } I}{\Gamma, x : X \vdash_O x \text{ has } I} \text{ (INH-X)}
\end{array}$$

Figure 6. FGJ semantics

This abstract class can be further refined to *final bind* T to a particular type:

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

These classes are related by subtyping: `IntList <: NumberList <: List`. Only classes where T is final bound can be non-abstract. The analogous definition of List using type properties follows:

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

NumberList and IntList can be written as follows:

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

However, note that our version of List is not abstract; T need not be instantiated by a subclass because it can be instantiated on a per-object basis in the constructor. There is no need to declared classes for NumberList and IntList. Instead, one can simply use the types `List{T<:Number}` and `List{T==Integer}`.

In addition, unlike virtual types, type properties can be refined contravariantly. For instance, one can write the type `List{T>Integer}`, and even `List{Integer<:T, T<:Number}`.

## 4.2 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 [24, 64] 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 to wildcards is achieved by leaving the element type property of Set unconstrained. We can thus 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:

$$\begin{aligned}
\llbracket X \rrbracket &= \_i \quad \text{where } X \text{ is the } i\text{th type parameter} \\
\llbracket C<?, \dots \rrbracket &= C\{\dots\} \\
\llbracket C<? \text{ extends } T, \dots \rrbracket &= C\{\_1 <: \llbracket T \rrbracket, \dots\} \\
\llbracket C<? \text{ super } T, \dots \rrbracket &= C\{\_1 >: \llbracket T \rrbracket, \dots\} \\
\llbracket C<T, \dots \rrbracket &= C\{\_1 == \llbracket T \rrbracket, \dots\}
\end{aligned}$$

Through such a translation, the FX(G)calculus in Section 3 can capture the essence of Java's wildcards, but extended with support for run-time type introspection.

Constrained types also support *proper abstraction* [64]. To illustrate, a reverse operation can operate on List of any type:

```
def reverse(list: List) = {
  for (i: int in [0..list.length-1]) {
    val tmp: list.T = list(i);
    list(i) = list(list.length-1-i);
    list(list.length-1-i) = tmp;
  }
}
```

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. Wildcard capture allows the parametrized method to be called with any List, regardless of its parameter type. 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.

## 4.3 Self types

Type properties can also be used to support a form of self types [8, 9]. Self types can be implemented by introducing a type property class to the root of the class hierarchy, Object:

```
class Object[class] { ... }
```

Scala's path-dependent types [45] and J&'s dependent classes [44] take a similar approach.

Self types are achieved by implicitly constraining types so that if an path expression p has type C, then `p.class <: C`. In particular, `this.class` is guaranteed to be a subtype of the lexically enclosing class; the type `this.class` represents all instances of the fixed, but statically unknown, run-time class referred to by the `this` parameter.

Self types address the binary method problem [8]. In the following example, the class BitSet can be written with a union method that takes a self type as argument.

```
interface Set {
  def union(s: this.class): void;
}

class BitSet implements Set {
```

```

    int bits;
    def union(s: this.class): void {
        this.bits |= s.bits;
    }
}

```

Since `s` has type `this.class`, and the class invariant of `BitSet` implies `this.class <: BitSet`, the implementation of the method is free to access the `bits` field of `s`.

Callers of `BitSet.union()` must call the method with an argument that has the same run-time class as the receiver. For a receiver `p`, the type of the actual argument of the call must have a constraint that entails `self.class==p.class`.

#### 4.4 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.

In this section, 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 HAS-F, METHOD-B, METHOD-I, and METHOD-C,E in Figure 4. The constraint system need change only to add structural constraints of the form “`T has I`” trivially derived from the existing constraints.

Structural constraints on types are found in many languages. Haskell supports type classes [32, 25]. 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 [42] allows type parameters to be bounded using structural *where clauses* [16]. For example, the sorted list class from Figure 3 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 def compareTo(T): int} {
    def add(x: T) = { ... x.compareTo(y) ... }
    ...
}

```

Structural method constraints permit the introduction of CLU-style optional methods [35]. 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`,

returning a `double`. However, `Array[Rabbit]`, for example, does not support the `add` operation because `Rabbit` does not have an `add` method.

## 5. Translation

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

The design is a hybrid design based on the implementation of parametrized classes in NextGen [11, 2, 3] and the implementation of PolyJ [42]. Generic classes are translated into template classes that are instantiated on demand at run time by binding the type properties to concrete types. To implement run-time type checking (e.g., casts), type properties are represented at run time using *adapter objects*.

This design, extended to handle language features not described in this paper, has been implemented in the X10 compiler. The X10 compiler is built on the Polyglot framework and translates X10 source to Java source<sup>9</sup>

### 5.1 Classes

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

For example, consider the following classes.

```

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

```

The compiler generates the following code:

```

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

    T a;

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

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

```

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

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

```

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

```

<sup>9</sup>There is also a translation from X10 to C++ source, not described here.

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

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

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

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

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

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

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

Could do instantiation eagerly, but quickly gets out of hand without whole-program analysis to limit the number of instantiations: 9 instantiations for one type property, 81 for two type properties, 729 for three.

Value constraints are erased from type references.

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

Consider the code in Figure 7. It contains most of the features of generics that have to be translated.

## 5.2 Eliminating method type parameters

## 5.3 Translation to Java

## 5.4 Run-time instantiation

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

## 6. 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 was developed by Mitchell [41] and by Reynolds [53]. These and subsequent systems are based on constraints over types, but not over values. Trifonov and Smith [65] proposed a type system in which types are refined using subtyping constraints. Pottier [49, 51] presents a constraint-based type system for an ML-like language with subtyping.

These developments lead to  $HM(X)$  [59, 50, 52], 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. Sulzmann and Stuckey [60] showed that the type inference algorithm for  $HM(X)$  can be encoded as a constraint logic program parametrized by the constraint system  $X$ . Type inference in this framework is thus constraint solving.

Structural constraints on types have been implemented in Haskell type classes [25]. PolyJ [42] also uses structural constraints to bound type parameters.

**Dependent types.** Dependent type systems [68, 14, 39, 6] parametrize types on values. Refinement type systems [23, 1, 33, 26, 20, 21, 54], introduced by Freeman and Pfenning [23], 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, [68], 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 [54], 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 [20, 21] 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  $FX(G)$ , dynamic type checks, including tests of dependent constraints, are inserted only at explicit casts or `instanceof` expressions; constraint solving is performed at compile time.

Concoction [22] extends types in OCaml [34] with constraints written as Coq [13] 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. Concoction 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, 47, 42, 62, 2]. GJ [7] implements invariant type parameters via type erasure. PolyJ [42] supports run-time representation of types via adapter objects, and

```

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

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

```

Figure 7. Code to translate

also permits instantiation of parameters on primitive types and structural parameter bounds. Viroli and Natali [67, 66] also support a run-time representation of types, using Java’s reflection API. NextGen [11, 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<sup>#</sup> also supports generic types via run-time instantiation in the CLR [61]. Type parameters may be declared with definition-site variance tags. Generalized type constraints were proposed for C<sup>#</sup> [17]. 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.

FX(G) does not support *bivariance* [29]; a class C is bivariant in a type property X if C{self.X==S} is a subtype of C{self.X==T} for any S and T. Bivariance is useful for writing code in which the property X is ignored. One can achieve this effect in FX(G) simply by leaving X unconstrained.

**Virtual classes and types.** *Virtual classes* [36, 37, 19]. are a language-based extensibility mechanism that were originally introduced in the language BETA [36] as a mechanism for supporting genericity. Virtual classes in BETA are not statically type safe, but this has been remedied in recent formulations [18, 19] and in variants of virtual classes [45, 44, 12, 30] using path-dependent types.

*Virtual types*, also introduced in BETA [36], 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 [62] proposed extending Java with virtual types as a genericity mechanism. Virtual types influenced Java’s wildcards [64, 24, 10].

Igarashi and Pierce [28] model the semantics of virtual types and several variants in a typed lambda-calculus with subtyping and dependent types.

Use-site variance based on structural virtual types were proposed by Thorup and Torgerson [63] and extended for parametrized type systems by Igarashi and Viroli [29]. The latter type system led to the development of wildcards in Java [24, 64, 10].

## 7. Conclusions

**todo:** rewrite

We have presented a preliminary design for supporting genericity in X10 using type properties. This type system generalizes the existing X10 type system. The use of constraints on type properties

allows the design to capture many features of generics in languages like Java 5 and C<sup>#</sup> and then to extend these features with new more expressive power. We expect that the design admits an efficient implementation and intend to implement the design shortly.

## 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 [48], 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 Programming Language. In *OOPSLA*, 1998.
- [8] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, Fall 1995.
- [9] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *LNCS*, pages 27–51. Springer-Verlag, 1995.
- [10] 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.

- [11] 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.
- [12] 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.
- [13] The Coq proof assistant: Reference manual, version 8.1. <http://coq.inria.fr/>, 2006.
- [14] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2–3), 1988.
- [15] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.
- [16] 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).
- [17] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for c# generics. In *ECOOP*, 2006.
- [18] 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.
- [19] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. pages 270–282, Charleston, South Carolina, January 2006.
- [20] Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd Annual Symposium on Principles of Programming Languages (POPL'06)*, pages 245–256, 2006.
- [21] 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.
- [22] Seth Fogarty, Emir Pašalić, Jeremy Siek, and Walid Taha. Concoction: 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.
- [23] 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.
- [24] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2006.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 161–185. Springer-Verlag, June 1999.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] Haskell 98: A non-strict, purely functional language. <http://www.haskell.org/onlinereport/>, February 1999.
- [33] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.
- [34] Xavier Leroy et al. The Objective Caml system. <http://caml.inria.fr/ocaml/>.
- [35] Barbara Liskov et al. *CLU Reference Manual*. Springer-Verlag, 1984.
- [36] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.
- [37] 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.
- [38] Per Martin-Löf. *A Theory of Types*. 1971.
- [39] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [40] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
- [41] 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.
- [42] 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.
- [43] Karl A. Nyberg, editor. *The annotated Ada reference manual*. Grebyn Corporation, Vienna, VA, USA, 1989.
- [44] 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.
- [45] Martin Odersky. Report on the programming language Scala. Technical report, EPFL, 2006.
- [46] 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)*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, July 2003.
- [47] 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.
- [48] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.
- [49] François Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, 1996.
- [50] François Pottier. A semi-syntactic soundness proof for HM(X). Technical Report RR 4150, INRIA, March 2001.
- [51] François Pottier. Simplifying subtyping constraints, a theory. *Information and Computation*, 170(2):153–183, November 2001.
- [52] François Pottier and Didier Rémy. *The Essence of ML Type Inference*, chapter 10. In Pierce [48], 2004.
- [53] John C. Reynolds. Three approaches to type structure. In *Proceedings*

of TAPSOFT/CAAP 1985, volume 185 of LNCS, pages 97–138. Springer-Verlag, 1985.

- [54] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.
- [55] 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.
- [56] Vijay Saraswat. The category of constraint systems is Cartesian closed. In *LICS '92*, pages 341–345, 1992.
- [57] Vijay Saraswat et al. The X10 language specification. Technical report, IBM T.J. Watson Research Center, 2006.
- [58] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR*, 2005.
- [59] 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.
- [60] Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming*, 18(2):251–283, 2008.
- [61] 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.
- [62] Kresten Krab Thorup. Genericity in Java with virtual types. Number 1241 in *Lecture Notes in Computer Science*, pages 444–471. Springer-Verlag, 1997.
- [63] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining virtual types and parameterized classes. In *ECOOP*, 1998.
- [64] 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.
- [65] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Third International Static Analysis Symposium (SAS)*, number 1145 in LNCS, pages 349–365, 1996.
- [66] Mirko Viroli. A type-passing approach for the implementation of parametric methods in java. *The Computer Journal*, 46(3):263–294, 2003.
- [67] 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.
- [68] 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.