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 $FX(C)$ 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

Large-scale systems present a number of challenges for modern object-oriented programming languages. Concurrency, distribution, and heterogeneous hardware and software environments require programming languages to support a rich set of features to be able to catch programmer errors at compile-time, to permit great expressiveness, and to enable code reuse.

X10 is a modern statically typed object-oriented language designed for high-performance concurrent and distributed programming [53]. A key data structure in X10 programs is the array: X10 supports dense, distributed, multi-dimensional arrays of objects, defined over sets of indices known as *regions*. To define arrays—and other data structures—in the X10 standard library, X10 supports both dependent types and generic types.

Generic types [31, 39, 7, 38, 57, 22, 56] enable more precise static type checking and allow programmers to reuse code in a va-

riety of contexts. Generic types are vital for implementing libraries, especially collections classes.

Dependent type systems [34, 63, 42, 5, 6, 4, 14] allow programmers to enforce run-time program invariants at compile-time. Dependent types in X10 are used to ensure the safety of array accesses. In addition, X10 uses dependent types to enforce X10's distribution model by ensuring remote data is not accessed as if it were local, and to enforce restrictions on concurrent accesses to prevent deadlock and data races.

The key idea is to define *constrained types*, a form of dependent type defined on predicates over types and over the immutable state of the program.

In X10, objects may have both value members (fields) and type members. The immutable state of an object is captured by its *value properties*: public final fields of the object. For instance, the following class declares a two-dimensional point with properties x and y of type int:

```
class Point2(x: int, y: int) { }
```

A constrained type in X10 is a type C{e}, where C is a class—called the *base class*—and e is a *constraint*, or list of constraints, on the properties of C and the final variables in scope at the type. For example, given the above class definition, the type Point2{x==y} is the type of all points on the diagonal.

Constraints on properties induce a natural subtyping relationship: C{c} is a subtype of D{d} if C is a subclass of D and c entails d. Thus, Point2{x==1,y==1} is a subtype of Point2{x>0}, which in turn is a subtype of Point2{true}—written simply as Point2.

In previous work [53, 51], we considered only value properties. In this paper, to support genericity these types are generalized to allow *type properties*, type-valued instance members of an object. Types may be defined by constraining the type properties as well as the value properties of a class.

The following code declares a class Array with a type property named T. The Array class maps points defined in a given region to values of type T.

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

Type properties are in many ways similar to type parameters as provided in object-oriented languages such as Java [22] and Scala [41] and in functional languages such as ML [36] and Haskell [29]. As the Array example illustrates, type properties are types in their own right: they may be used in any context a type may be used, including in instanceof and cast expressions.

However, the key distinction between type properties and type parameters is that type properties are instance members. Thus, for an expression a of type Array, a.T is a type, equivalent to the concrete type to which T was initialized when the object a was

instantiated. To ensure soundness, `a` is restricted to final access paths. Within the body of a class, the unqualified property name T resolves to `this.T`.

All properties of an object, both type and value, must be bound at object instantiation and are immutable once bound. Thus, the type property `T` of a given `Array` instance must be bound by the constructor to a concrete type such as `String` or `Point2{x>=0}`.

As with value properties, type properties may be constrained by predicates to produce *constrained types*. Many features of modern object-oriented type systems fall out naturally from this type system.

X10 supports equality constraints, written $T_1 == T_2$, and subtyping constraints, written $T_1 <: T_2$, on types. For instance, the type `Array{T==float}` is the type of all `Arrays` containing a `float`. For an instance `a` of this type, the types `a.T` and `float` are equivalent.

Subtyping constraints enable *use-site variance* [27]. The type `Array{T<:Collection}` constrains T such that all instances of the type must bind T to a subtype of `Collection`. Variables of this type may contain `Arrays` of `Collection`, `Array` of `List`, or `Array` of `Set`, etc.

Constraints on value properties may be used to ensure array bounds are not violated. Arrays in X10 are distributed over a number of abstract *places* (which the language runtime maps to physical nodes in a distributed system). The property `dist` defines a mapping from a point `p` to the place where the storage for the array element at `p` is located. The index points given to the `get` and `set` methods have constrained types that ensure the array is accessed only using points in the region on which the array is defined.

The following code declares a version of `Array` with a type property named T and value properties `dist` of type `Dist` and `region` of type `Region`. The region of the array is constrained to be the same as the region of the distribution.

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

Our previous work [51] was a first step toward defining a dependent type system for an object-oriented language as a constraint system. In this paper, we present a constraint-based framework for formalizing object-oriented type systems. The type system is parametrized on a constraint system. By augmenting the default constraint system, the type system can serve as a core calculus for formalization of extensions of a core object-oriented language.

The same framework can be extended to support generic types with structural rather than nominal bounds [38, 23], to model other features of object-oriented languages such as virtual types [33, 32, 17] and self types [8, 9].

Contributions. We present a constraint-based framework FX() for formalizing object-oriented type systems. The framework is based on Featherweight Java [25] and is parametrized on a constraint system over types and values. By augmenting the default constraint system, the type system can serve as a core calculus for formalization of extensions of a core object-oriented language:

- FX(G) supports generic types,
- FX(D($\mathcal{A}$)) supports dependent types,
- FX(G,D($\mathcal{A}$)) supports both generics and dependent types.

The formal system has been proved sound.

The constraint-based framework forms the basis for a practical, implementable language. We describe the implementation of an extension of X10 that provides both generic and dependent types. 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.

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 $t_1, \ldots, t_n$; terms with list subterms are considered a single list of terms (e.g., we write $\bar{x}:\bar{T}$ for the list $x_1:T_1, \ldots, x_n:T_n$).

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 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.

```
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

```
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

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.

2.2 Field, method, and constructor constraints

Field, method, and constructor 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 the `this.length>0` to access `head`; that is `this.head` may not be dereferenced unless `this` has type `List{length>0}`. Similarly, `tail` cannot be accessed unless the tail is non-empty. 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 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*. A method with a method constraint is called a *conditional method*. The method constraint on `get` requires that the actual receiver's `length` field must be non-zero—calls to `get` on empty lists are not permitted. This constraint 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 the `tail`. Note that for this example to type-check, the constraint system must support conditional constraints to guarantee that `i` is within the loop bounds of `tail`. The constraint solver must prove

$$\text{length} > 0, \text{tail.length} = \text{length} - 1, 0 \le \text{i}, \text{i} < \text{length}, \text{i} \ne 0$$
$$\vdash \text{tail.length} > 0, 0 \le \text{i} - 1, \text{i} - 1 < \text{tail.length}$$

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`).[1]

`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 [27].

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`.

---

[1] 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{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 $O(\mathcal{X})$ on top of $\mathcal{X}$ which captures constraints related to the object-oriented structure of P. O is sensitive to $\mathcal{X}$ only in that 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 $O(\mathcal{X})$.

3.1 The Object constraint system, O

Given $\mathcal{L}$, its input constraint system $\mathcal{X}$ we now show how to define O. The inference relation for 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 O. Thus we should think of $\mathcal{X}$ and O as being defined simultaneously and recursively.

The constraints of O are given by:

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

class(C) is intended to be true for all classes `C` defined in the program. $\text{S} \trianglelefteq \text{T}$ 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. $\text{S} \trianglelefteq \text{T}$ 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.

O satisfies the following axioms and inference rules in Figure 4. Since O is a constraint system [52], 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 $\text{C}<:\text{D}$ and C has I, for classes `C`, `D` and members `I`; that is, if the rules cannot be used to establish $\vdash_O \text{C}<:\text{D}$ ($\vdash_O \text{C has I}$), then it is the case that $\vdash_O \text{C} \not<: \text{D}$ ($\vdash_O \neg(\text{C has I})$).

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).

3.2 Judgments

In the following Γ 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 ϕ in the sequence all variables `x` (`X`) occurring in ϕ are defined by a declaration `x : T` (`X type`) in the sequence to the left of ϕ.

2. for any variable `x` (`X`), there is at most one formula `x : T` (`X type`) in Γ.

The judgments of interest are as follows. (1) Type well-formedness: $\Gamma \vdash \text{T type}$, (2) Subtyping: $\Gamma \vdash \text{S}<:\text{T}$, (3) Typing: $\Gamma \vdash \text{e} : \text{T}$, (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: $\Gamma \vdash \text{Cl}$ *OK* (class definition `Cl` 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 allows.

$$\frac{\texttt{class C}(\overline{\texttt{f}}:\overline{\texttt{V}})\texttt{ extends D}\ldots\in P}{\vdash_O \texttt{class(C)},\texttt{C}\trianglelefteq\texttt{D}} \quad (\textsc{Class}) \qquad\qquad \frac{\Gamma\vdash_O\texttt{fields(x)}=\overline{\texttt{f}}:\overline{\texttt{V}}}{\Gamma\vdash_O\texttt{x has f}_i:\texttt{V}_i} \quad (\textsc{Has-F})$$

$$\vdash_O\texttt{new D}(\overline{\texttt{t}}).\texttt{f}_i=\texttt{t}_i \quad (\textsc{Sel}) \qquad\qquad \frac{\Gamma\vdash_O\texttt{x}:\texttt{C},\texttt{class(C)}}{\Gamma\vdash_O inv(\texttt{C},\texttt{x})} \quad (\textsc{Inv})$$

$$\vdash_O T\trianglelefteq T \quad (\textsc{V-Id}) \qquad \frac{\Gamma\vdash_O\texttt{t}\trianglelefteq\texttt{T}}{\Gamma\vdash_O\texttt{t}<:\texttt{T}} \quad (\textsc{Sub-X}) \qquad \frac{\Gamma\vdash_O\texttt{T}_1<:\texttt{T}_2,\texttt{T}_2<:\texttt{T}_3}{\Gamma\vdash_O\texttt{T}_1<:\texttt{T}_3} \quad (\textsc{S-Trans})$$

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

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

$$x:\texttt{Object}\vdash_O\texttt{fields(x)}=\bullet \quad (\textsc{Fields-B}) \qquad \frac{\Gamma,\texttt{x}:\texttt{D}\vdash_O\texttt{fields(x)}=\overline{\texttt{g}}:\overline{\texttt{V}}\quad\texttt{class C}(\overline{\texttt{f}}:\overline{\texttt{U}})\{\texttt{c}\}\texttt{ extends D}\{\overline{\texttt{M}}\}\in C}{\Gamma,\texttt{x}:\texttt{C}\vdash\texttt{fields(x)}=\overline{\texttt{g}}:\overline{\texttt{V}},\overline{\texttt{f}}:\overline{\texttt{U}}[\texttt{x}/\texttt{this}]} \quad (\textsc{Fields-I})$$

$$\frac{\Gamma,\texttt{x}:\texttt{S}\vdash_O\texttt{fields(x)}=\overline{\texttt{f}}:\overline{\texttt{V}}}{\Gamma,\texttt{x}:\texttt{S}\{\texttt{d}\}\vdash_O\texttt{fields(x)}=\overline{\texttt{f}}:\overline{\texttt{V}\{\texttt{d}[\texttt{x}/\texttt{self}]\}}}$$
$$\Gamma,\texttt{x}:(\texttt{y}:\texttt{U};\texttt{S})\vdash_O\texttt{fields(x)}=\overline{\texttt{f}}:\overline{(\texttt{y}:\texttt{U};\texttt{V})} \quad (\textsc{Fields-C,E})$$

$$\frac{\Gamma,\texttt{x}:\texttt{C}\vdash_O\texttt{class(C)}\quad\theta=[\texttt{x}/\texttt{this}]\quad\texttt{def m}(\overline{\texttt{z}}:\overline{\texttt{V}})\{\texttt{c}\}:\texttt{T}=\texttt{e}\in P}{\Gamma,\texttt{x}:\texttt{C}\vdash_O\texttt{x has }(\texttt{m}(\overline{\texttt{z}}:\overline{\texttt{V}}\theta)\{\texttt{c}\theta\}:\texttt{T}\theta=\texttt{e})} \quad (\textsc{Method-B})$$

$$\frac{\Gamma,\texttt{x}:\texttt{D}\vdash_O\texttt{x has m}(\overline{\texttt{z}}:\overline{\texttt{V}})\{\texttt{c}\}:\texttt{T}=\texttt{e}\quad\texttt{class C}(\ldots)\texttt{ extends D}\{\overline{\texttt{M}}\}\quad\texttt{m}\notin\overline{\texttt{M}}}{\Gamma,\texttt{x}:\texttt{C}\vdash_O\texttt{x has m}(\overline{\texttt{z}}:\overline{\texttt{V}})\{\texttt{c}\}:\texttt{T}=\texttt{e}} \quad (\textsc{Method-I})$$

$$\frac{\Gamma,\texttt{x}:\texttt{S}\vdash_O\texttt{x has m}(\overline{\texttt{z}}:\overline{\texttt{V}})\{\texttt{c}\}:\texttt{T}=\texttt{e}}{\Gamma,\texttt{x}:\texttt{S}\{\texttt{d}\}\vdash_O\texttt{x has m}(\overline{\texttt{z}}:\overline{\texttt{V}})\{\texttt{c}\}:\texttt{T}\{\texttt{d}[\texttt{x}/\texttt{self}]\}=\texttt{e}}$$
$$\Gamma,\texttt{x}:(\texttt{y}:\texttt{U};\texttt{S})\vdash_O\texttt{x has m}(\overline{\texttt{z}}:\overline{\texttt{V}})\{\texttt{c}\}:(\texttt{y}:\texttt{U};\texttt{T})=\texttt{e} \quad (\textsc{Method-C,E})$$

Figure 4. The Object constraint system, O

$$\sigma(\varepsilon)=\texttt{true}$$
$$\sigma(\texttt{x}:\texttt{T}\{\texttt{c}\},\Gamma)=\texttt{c}[\texttt{x}/\texttt{self}],\sigma(\Gamma)$$
$$\sigma(\texttt{x}:(\texttt{y}:\texttt{S};\texttt{T}),\Gamma)=\sigma(\texttt{y}:\texttt{S},\texttt{x}:\texttt{T},\Gamma)$$
$$\sigma(c,\Gamma)=c,\sigma(\Gamma)$$

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 Γ.

We say that a context Γ 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 Γ is consistent; if it is inconsistent, the rule cannot be used and the type of the given expression cannot be established (type-checking fails).

3.3 FX()

The semantics of 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 $\{\texttt{c}\}$. 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.[2]

The syntax for constraints in FX() is specified in Figure 5. 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 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.

---

[2] 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.

FX productions:

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

(Type)	S,T	$::=$	$N \mid T\{c\} \mid (x{:}S;T)$
(N Type)	N	$::=$	$C \mid N\{c\}$
(C Term)	t	$::=$	$x \mid \text{self} \mid \text{this} \mid t.f \mid \text{new } C(\overline{t})$
(Const.)	c,d	$::=$	$\text{true} \mid t{==}t \mid c,c \mid x{:}V;c$

FX well-formedness rules:

$$\Gamma \vdash \text{true} : o \qquad (\textsc{True})$$

$$\frac{\Gamma \vdash c_0 : o \qquad \Gamma \vdash c_1 : o}{\Gamma \vdash (c_0, c_1) : o} \quad (\textsc{And})$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash c[t/x] : o}{\Gamma \vdash x : T; c : o} \quad (\textsc{Exists})$$

$$\frac{\Gamma \vdash \text{class}(C)}{\Gamma \vdash C \text{ type}} \quad (\textsc{Class})$$

$$\frac{\Gamma \vdash S \text{ type}, T \text{ type}}{\Gamma \vdash x : S; T \text{ type}} \quad (\textsc{Exist-T})$$

$$\frac{\Gamma \vdash T \text{ type} \qquad \Gamma, \text{self} : T \vdash c : o}{\Gamma \vdash T\{c\} \text{ type}} \quad (\textsc{Dep})$$

Type judgment rules:

$$\Gamma, x : T \vdash x : T\{\text{self} == x\} \quad (\textsc{T-Var})$$

$$\frac{\Gamma \vdash e : U \qquad \Gamma \vdash T \text{ type}}{\Gamma \vdash e \text{ as } T : T} \quad (\textsc{T-Cast})$$

$$\frac{\Gamma \vdash e : S \qquad \Gamma, z : S \vdash z \text{ has } f : V \qquad (z \text{ fresh})}{\Gamma \vdash e.f : (z : S; V\{\text{self} = z.f\})} \quad (\textsc{T-Field})$$

$$\frac{\begin{array}{c} \Gamma \vdash e : T, \overline{e} : \overline{V} \\ \Gamma, v : T, \overline{v} : \overline{V} \vdash v \text{ has } (m(\overline{v}:\overline{U}), c \to S), \overline{V} <: \overline{U}, c \qquad (v, \overline{v} \text{ fresh}) \end{array}}{\Gamma \vdash e.m(\overline{e}) : (v : T; \overline{v} : \overline{V}; S)} \quad (\textsc{T-Invk})$$

$$\frac{\begin{array}{c} \Gamma \vdash \overline{e} : \overline{T} \qquad \vdash \text{class}(C) \\ \Gamma, v : C \vdash \text{fields}(v) = \overline{f} : \overline{V} \qquad (v, \overline{v} \text{ fresh}) \\ \Gamma, v : C, \overline{v} : \overline{T}, v.\overline{f} = \overline{v} \vdash \overline{T} <: \overline{V}, \mathit{inv}(C, v) \end{array}}{\Gamma \vdash \text{new } C(\overline{e}) : C\{\overline{v} : \overline{T}; \text{new } C(\overline{v}) = \text{self}, \mathit{inv}(C, \text{self})\}} \quad (\textsc{T-New})$$

$$\frac{\text{this} : C \vdash c : o \qquad \text{this} : C, \overline{x} : \overline{V}, c \vdash T \text{ type}, \overline{V} \text{ type}, e : S, S <: T}{\text{def } m(\overline{x}:\overline{V})\{c\} : T = e; \text{ OK in } C} \quad (\textsc{Method OK})$$

$$\frac{\overline{M} \text{ OK in } C \qquad \text{this} : C \vdash c : o \qquad \text{this} : C, c \vdash \overline{V} \text{ type}, N \text{ type}}{\text{class } C(\overline{f}:\overline{V})\{c\} \text{ extends } N\{\overline{M}\} \text{ OK}} \quad (\textsc{Class OK})$$

Transition rules:

$$\frac{x : C \vdash \text{fields}(x) = \overline{f} : \overline{V}}{(\text{new } C(\overline{e})).f_i \to e_i} \quad (\textsc{R-Field})$$

$$\frac{x : C \vdash x \text{ has } m(\overline{x} : \overline{T})\{c\} : T = e}{(\text{new } C(\overline{e})).m(\overline{d}) \to e[\text{new } C(\overline{e}), \overline{d}/\text{this}, \overline{x}]} \quad (\textsc{R-Invk})$$

$$\frac{e \to e'}{e.f_i \to e'.f_i} \quad (\textsc{RC-Field})$$

$$\frac{e \to e'}{e.m(\overline{e}) \to e'.m(\overline{e})} \quad (\textsc{RC-Invk-Recv})$$

$$\frac{\vdash C\{\text{self} == \text{new } C(\overline{d})\} <: T}{(T)(\text{new } C(\overline{d})) \to \text{new } C(\overline{d})} \quad (\textsc{R-Cast})$$

$$\frac{e_i \to e_i'}{e.m(\ldots, e_i, \ldots) \to e.m(\ldots, e_i', \ldots)} \quad (\textsc{RC-Invk-Arg})$$

$$\frac{e \to e'}{(T)e \to (T)e'} \quad (\textsc{RC-Cast})$$

$$\frac{e_i \to e_i'}{\text{new } C(\ldots, e_i, \ldots) \to \text{new } C(\ldots, e_i', \ldots)} \quad (\textsc{RC-New-Arg})$$

Figure 5. Semantics of FX

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][3]; 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 "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[4], 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}).

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($\overline{v}$) for some values $\overline{v}$ 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.

3.4 FX(G)

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

FX(G) is the language obtained by adding the following productions to FX().

(Par Type)	V	::=	type
(Path)	p	::=	x | self | this | p.f
(Type)	T	::=	X | p
(C Term)	t	::=	T
(Const.)	c	::=	t $\trianglelefteq$ N | t==t

That is, first we introduce the "type" type. FGJ method type parameters are modeled in FX(G) as normal parameters of type

type.[5] 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 **??**. 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 [41]). 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 $t \trianglelefteq N$. It may be used, for instance, to specify upper bounds on type variables or fields (path types).[6]

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 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($\overline{t}$)
(Const.)	c	::=	q($\overline{t}$)

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

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

$$\frac{f(\overline{T}) : T \in \mathcal{C} \qquad \Gamma \vdash \overline{t} : \overline{T}}{\Gamma \vdash f(\overline{T}) : T} \tag{FUN}$$

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

No additional type judgments or transition rules are needed.

---

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

[4] Note from the definition of fields in O (Figure 4) that all occurrences of this in the declared type of the field f will have been replaced by z.

[5] 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.

[6] To support structural typing, we permit the programmer to use x hasI constraints, see Section **??**.

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

$$\frac{\Gamma \vdash_O p \trianglelefteq T \quad \Gamma, x : T \vdash_O x \text{ has } I}{\Gamma, x : p \vdash_O x \text{ has } I} \text{ (Inh-P)} \qquad \frac{\Gamma \vdash_O X \trianglelefteq T \quad \Gamma, x : T \vdash_O x \text{ has } I}{\Gamma, x : X \vdash_O x \text{ has } I} \text{ (Inh-X)}$$

Figure 6. FGJ semantics

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

No additional rules are needed beyond those of FX(G) and 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 FX(G,D($\mathcal{A}$)).

THEOREM 3.1 (Subject Reduction). *If* $\Gamma \vdash e : T$ *and* $e \to e'$ *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:*

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

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

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.* $e \to e'$.

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 [33, 32, 16, 17, 12] and similar constructs built on path-dependent types found in languages such as Scala [41], and J& [40]. Indeed, one of the first proposals for adding genericity to Java was via 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, **?**, 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;
  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;
}
```

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 is as 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 [22, 59] can be motivated by the following example in Java. Consider a Set<T> class with static field EMPTY containing an empty collection. What type should EMPTY be?

```
class Set<T> {
  public static final 〈whattype?〉 EMPTY = ...;
  ...
}
```

One option is to make the type of `EMPTY` be `Set<T>`. However, then `Set<String>.EMPTY` and `Set<Float>.EMPTY` are distinct variables. We would like to share `EMPTY` across all classes. The solution in Java is to use a wildcard:

```
class Set<T> {
  public static final Set<?> EMPTY = ...;
  ...
}
```

`EMPTY`'s type is the type of collections of *some* parameter T. In X10, a similar effect is achieved by leaving the type property of `Set` unconstrained:

```
class Set[T] {
  const EMPTY: Set;
  ...
}
```

We can thus define the following straightforward translation from Java wildcard types to X10 types:

$$\llbracket C<?> \rrbracket = C$$
$$\llbracket C<? \text{ extends } Foo> \rrbracket = C\{T <: Foo\}$$
$$\llbracket C<? \text{ extends } Bar> \rrbracket = C\{T :> Bar\}$$
$$\llbracket C<Baz> \rrbracket = C\{T == Baz\}$$

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]) {
    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 [41] and J&'s dependent classes [40] 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 `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 [29, 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 [38] allows type parameters to be bounded using structural *where clauses*. For example, the sorted list class from Figure 3 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) ... }
  ...
}
```

Structural method constraints permit the introduction of CLU-style optional methods [31]. 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. 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 [38]. 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[7]

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 used in place of a constructor call. This code is compiled to Java bytecode.

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

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

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

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

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

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

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

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

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 [37] and by Reynolds [49]. 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 [45, 47] presents a constraint-based type system for an ML-like language with subtyping.

These developments lead to HM(X) [54, 46, 48], 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 [55]

---

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

```
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

---

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 [23]. PolyJ [38] also uses structural constraints to bound type parameters.

Dependent types. Dependent type systems [63, **?**, 35, 6] parametrize types on values. Refinement type systems [21, 1, 30, 24, 18, 19, 50], 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 [50], 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(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 [**?**] 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. 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, 43, 38, 57, 2]. GJ [7] implements invariant type parameters via type erasure. PolyJ [38] 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 [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.

$C^\sharp$ 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 $C^\sharp$ [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.

FX(G) does not support *bivariance* [27]; 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* [32, 33, 17]. are a language-based extensibility mechanism that where originally in-

troduced in the language BETA [32] 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 [41, 40, 12, 28] using path-dependent types.

Virtual types, also introduced in BETA [32], 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, 10].

Igarashi and Pierce [26] 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 [58] and extended for parametrized type systems by Igarashi and Viroli [27]. The latter type system lead to the development of wildcards in Java [22, 59, 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$^\sharp$ 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

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 [44], 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] 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, 1988.

[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. 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 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.

[27] 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.

[28] 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.

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

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

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

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

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

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

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

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

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

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

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

[46] François Pottier. A semi-syntactic soundness proof for HM(X). Technical Report RR 4150, INRIA, March 2001.

[47] François Pottier. Simplifying subtyping constraints, a theory. *Information and Computation*, 170(2):153–183, November 2001.

[48] François Pottier and Didier Rémy. *The Essence of ML Type Inference*, chapter 10. In Pierce [44], 2004.

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

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

[51] 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.

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

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

[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. 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.