Genericity through Dependently Constrained Types

Nathaniel Nystrom *
nystrom@us.ibm.com

Igor Peshansky *
igorp@us.ibm.com

Vijay Saraswat *
vsaraswa@us.ibm.com

Abstract

We present a general framework for *generic constrained types* that captures the notion of value-dependent and type-dependent (i.e., generic) type systems for object-oriented languages. Constraint systems formalize systems of partial information. Constrained types are formulas `C{c}` where `C` is the name of a class or an interface and `c` is a constraint on the immutable state of an instance of `C` (the *properties*).

The basic idea is to formalize the essence of nominal object-oriented types as a constraint system, and to permit both value and type properties and parameters. Type-generic dependence is now expressed through constraints on these properties and parameters. Type-valued properties are required to have a run-time representation—the run-time semantics is not defined through erasure.

Many type systems for object-oriented languages developed over the last decade can be thought of as constrained type systems in this formulation. This framework is parametrized by an arbitrary constraint system C of interest. It permits the development of languages with pluggable type systems, and supports dynamic code generation to check casts at run-time.

The paper makes the following contributions: (1) We show how to accommodate generic object-oriented types within the framework of constrained types. (1) We illustrate the type system with the development of a formal calculus GFX and establish type soundness. (3) We discuss the design and implementation of the type system for X10, a modern object-oriented language, based on constrained types. The type system integrates and extends the features of nominal types, virtual types, and Scala's path-dependent types, as well as representing generic types.

1. Introduction

```
Outline

Problem statement
X10 ... new language design
OO, imperative, concurrent
catch errors -> types
dependent types, generics
        practical, pluggable

=> constraint-based framework
  -> rich history in type
        OOPSLA: first step dep type system as a constraint system
in this paper we show,
1. extend constraint system applied to handle generics
combined wth dependent types
2. object type checking FJ style can be cast as a
constraint system
3. FGJ-style simply same constraint system but permitting
type variables (vars at type type)
4. can combine these two -- arrays need both!!

5. practical real language, implementable:
type-passing LM or PolyJ style
        and NextGen style type instantiation
   - design issues
   - user-defineable version (later)

beautiful small programs
        - list
        - gcd
        - sorted list
        - sorted binary tree?

class SortedList[T]{tl == null || hd < tl.hd} {
    hd: T;
    tl: SortedList[T];

    def this[T](hd: T, tl: SortedList[T]) {
        ...
    }

    def insert(x: T): SortedList[T]{hd == min(this.hd, x)} = {
        if (x < hd)
            return new SortedList[T](x, this);
        else
            return new SortedList[T](hd, tl.insert(x));
    }
}
```

todo: Awkward, repetitive

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

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

todo: Incorporate some text from OOPSLA paper on deptypes.

todo: Cite liquid types and whatever it cites

Modern object-oriented type systems provide many features to improve productivity by allowing programmers to express program invariants as types that are checked by the compiler, without sacrificing the ability to reuse code. We present a dependent type system that extends a class-based language with statically-enforced constraints on types and values. This type system supports several features of modern object-oriented language through natural ex-

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

tensions of the core dependent type system: generic types, virtual types, and self types among them.

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. The type system is parametrized on a constraint system. We have formalized the type system in an extension of Featherweight Java [22] and provide proof of soundness. By augmenting the default constraint system, the type system can serve as a core calculus for formalizing extensions of a core object-oriented language.

This work is done in the context of the X10 programming language [45]. 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 `float`:

```
class Point(x: float, y: float) { }
```

A constrained type 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 `Point{x*x+y*y<1}` is the type of all points within the unit circle.

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, `Point{x==1,y==1}` is a subtype of `Point{x>0}`, which in turn is a subtype of `Point{true}`—written simply as `Point`.

In previous work [45, 43], 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 `Cell` with a type property named `T`.

```
class Cell[T] {
    var value: T;
    def get(): T = value;
    def set(v: T) = { value = v; }
}
```

The class has a mutable field `value` of type `T`, and has `get` and `set` methods for accessing the field.

This example shows that type properties are in many ways similar to type parameters as provided in object-oriented languages such as Java [19] and Scala [35] and in functional languages such as ML [31] and Haskell [26].

As the 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 `e` of type `Cell`, `e.T` is a type, equivalent to the concrete type to which `T` was initialized when the object `e` was instantiated. To ensure soundness, `e` is restricted to final access paths. Within the body of a class, 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 `Cell` instance must be bound by the constructor to a concrete type such as `String` or `Point{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.

Generic types. Constraints on type properties lead directly to a form of generic class. The `Cell` defined class above is a generic class. X10 supports equality constraints, written T_1`==`T_2, and subtyping constraints, written T_1`<:`T_2, on types. For instance, the type `Cell{T==float}` is the type of all `Cell`s containing a `float`. For an instance `c` of this type, the types `c.T` and `float` are equivalent. Thus, the following code is legal.

```
val x: float = c.get();
c.set(1.0);
```

Subtyping constraints enable *use-site variance* [**?**]. The type `Cell{T<:Collection}` constrains `T` to be a subtype of `Collection`. All instances with this type must bind `T` to a subtype of `Collection`. Variables of this type may contain `Cell`s of `Collection`, `Cell` of `List`, or `Cell` of `Set`, etc. Subtyping constraints provide similar expressive power as Java wildcards. We describe an encoding of wildcards in Section **??**.

Other type systems. While this paper focuses mainly on supporting generic types through dependent types, the formalism generalizes other object-oriented type systems, e.g., self types [6, 7], virtual types [29, 28, 14], and structually constrained types [33, 20].

1.1 Contributions

todo: We need some!

1.2 Implementation

Type properties are a powerful mechanism for providing genericity in X10. Unlike existing existing proposals for generic types in Java-like languages [19, 5, 36, 33, 49, 35], which are implemented via type erasure, our design supports run-time introspection of generic types.

Another problem with many of these proposals is inadequate support for primitive types, especially arrays. The performance of primitive arrays is critical for the high-performance applications for which X10 is intended. These proposals introduce unnecessary boxing and unboxing of primitives. Our design does not require primitives be boxed.

Outline. The rest of the paper is organized as follows. Section 6 discusses related work. An informal overview of generic constrained types in X10 is presented in Section 2. Section 3 presents a formal semantics and a proof of soundness. The implementation of generics in X10 by translation to Java is described in Section 5. Finally, Section 7 concludes.

todo: Fix this

2. X10 language overview

This section presents an informal description of generic constrained types in X10. The type system is formalized in a simplified version of X10, GFX (Generic Featherweight X10), in Section 3.

When presenting syntax, we follow the usual conventions for Featherweight Java: we write $\overline{\mathtt{t}}$ for the list $\mathtt{t}_1,\ldots,\mathtt{t}_n$; terms with list subterms are considered a single list of terms (e.g., we write $\overline{\mathtt{x}:\mathtt{T}}$ for the list $\mathtt{x}_1:\mathtt{T}_1,\ldots,\mathtt{x}_n:\mathtt{T}_n$).

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.

```
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

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 $\texttt{C[}\overline{\texttt{T}}\texttt{](}\overline{\texttt{e}}\texttt{)}$ is shorthand for $\texttt{C[}\overline{\texttt{X}}\texttt{==}\overline{\texttt{T}}\texttt{](}\overline{\texttt{x}}\texttt{==}\overline{\texttt{e}}\texttt{)}$ where $\texttt{X}_i$ are the type properties and $\texttt{x}_i$ 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.

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.

```
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} = {
    if (i == 0) return head;
    return tail.get(i-1);
  }

  def map[S](f: (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

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 the type-checker must be flow sensitive to be able go guarantee that `i` is within the loop bounds of `tail`. The constraint solver must prove

$$\texttt{length} > 0, \texttt{tail.length} = \texttt{length} - 1, 0 \leq \texttt{i}, \texttt{i} < \texttt{length}, \texttt{i} \neq 0 \vdash \texttt{tail.length} > 0, 0 \leq \texttt{i} - 1, \texttt{i} - 1 < \texttt{tail.length}$$

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 `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 [24].

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 interchangable 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));
```

[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

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:

(Member)	`I`	::=	$\texttt{m}(\overline{\texttt{x}}:\overline{\texttt{V}})\{\texttt{c}\}:\texttt{T}=\texttt{e}$ | `f:V`
(Const.)	`c,d`	::=	`class(C)` | `S E T` | `S <: T`
			| $\texttt{fields}(\texttt{x})=\overline{\texttt{f}}:\overline{\texttt{V}}$
			| `x has I`

`class(C)` is intended to be true for all classes `C` defined in the program. `S E 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 E T` is intended to hold if `S` is a subtype of `T`. For a variable `x`, `fields(x)` is intended to specify the (complete) set of typed fields available to `x`. `x has I` is intended to specify that the member `I` (field or method) is available to `x` – for instance it is defined at the class at which `x` is declared or inherited by it, or it is available at the upper bound of a type variable.

$\mathcal{O}$ satisfies the following axioms and inference rules in Figure 4. Since $\mathcal{O}$ is a constraint system [44], it also satisfies Identity and Cut.

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

We assume that the rules given are complete for defining the predicates `C<:D` and `C has I`, for classes `C`,`D` and members `I`. That

is, if the rules cannot be used to establish $\vdash_O$ `C<:D` ($\vdash_O$ `C has I`), then it is the case that $\vdash_O$ `C` $\not<:$ `D` ($\vdash_O \neg$(`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 Judgements

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`) occuring 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 judgements 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: $\Gamma \vdash$ `Cl` *OK* (class definition `Cl` is ok).

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

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

[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.

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 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 judgement 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 runtime `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 runtime 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`($\overline{\mathtt{v}}$) for some values $\overline{\mathtt{v}}$ of the given type.

Operational semantics. The operational semantics is straightforward and essentially identical to FJ[22]. 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.

[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`.

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

$$\vdash_O \texttt{new D}(\overline{\texttt{t}})\texttt{.f}_\texttt{i} = \texttt{t}_\texttt{i}\ (\textsc{Sel}) \qquad \frac{\Gamma \vdash_O \texttt{x} : \texttt{C}; \texttt{class(C)}}{\Gamma \vdash_O \mathit{inv}(\texttt{C};\texttt{x})}\ (\texttt{Inv})$$

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

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

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

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

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

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

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

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

Figure 4. The Object constraint system, O

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 E 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 [35]). 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 E N. It may be used, for instance, to specify upper bounds on type variables or fields (path types).[6]

The following well-formedness and $\vdash_O$ rules must be added:

$$\frac{\Gamma \vdash \texttt{p} : \texttt{T} \qquad \Gamma;\texttt{x} : \texttt{T} \vdash \texttt{x has X} : \texttt{type}}{\Gamma \vdash \texttt{p.X type}}\ (\textsc{Path})$$

$$\texttt{S} == \texttt{T} \vdash_O S \trianglelefteq T; T \trianglelefteq S\ (\texttt{Equals})$$

$$\frac{\Gamma \vdash_O \texttt{p} \trianglelefteq \texttt{T} \qquad \Gamma;\texttt{x} : \texttt{T} \vdash_O \texttt{x has I}}{\Gamma;\texttt{x} : \texttt{p} \vdash_O \texttt{x has I}}\ (\texttt{Inh-p})$$

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

[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 **??**.

FX **productions:**

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

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

FX **well-formedness rules:**

$$\Gamma \vdash \mathtt{true} : o \quad (\mathtt{true}) \qquad \frac{\Gamma \vdash c_0 : o \qquad \Gamma \vdash c_1 : o}{\Gamma \vdash (c_0, c_1) : o}\ (\textsc{And}) \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash c[t{=}x] : o}{\Gamma \vdash x : T; c : o}\ (\textsc{Exists}) \qquad \frac{\Gamma \vdash \mathtt{class}(C)}{\Gamma \vdash C\ \mathtt{type}}\ (\textsc{Class})$$

$$\frac{\Gamma \vdash S\ \mathtt{type}, T\ \mathtt{type}}{\Gamma \vdash x : S; T\ \mathtt{type}}\ (\textsc{Exist-T}) \qquad \frac{\Gamma \vdash T\ \mathtt{type} \qquad \Gamma, \mathtt{self} : T \vdash c : o}{\Gamma \vdash T\{c\}\ \mathtt{type}}\ (\textsc{Dep})$$

Type judgement rules:

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

$$\frac{\Gamma \vdash e : T, \overline{e} : \overline{V} \qquad \Gamma, v : T, \overline{v} : \overline{V} \vdash v\ \mathtt{has}\ (m(\overline{v} : \overline{U}); c \to S); \overline{V} <: \overline{U}; c \qquad (v, \overline{v} \text{ fresh})}{\Gamma \vdash e.m(\overline{e}) : (v : T; \overline{v} : \overline{V}; S)}\ (\text{T-INVK})$$

$$\frac{\Gamma \vdash \overline{e} : \overline{T} \qquad \vdash \mathtt{class}(C) \qquad \Gamma, v : C \vdash \mathtt{fields}(v) = \overline{f} : \overline{V} \qquad (v, \overline{v} \text{ fresh}) \qquad \Gamma, v : C; \overline{v} : \overline{T}; v.\overline{f} = \overline{v} \vdash \overline{T} <: \overline{V}; \mathit{inv}(C; v)}{\Gamma \vdash \mathtt{new}\ C(\overline{e}) : C\{\overline{v} : \overline{T}; \mathtt{new}\ C(\overline{v}) = \mathtt{self}; \mathit{inv}(C; \mathtt{self})\}}\ (\text{T-NEW})$$

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

Transition Rules:

$$\frac{x : C \vdash \mathtt{fields}(x) = \overline{f} : \overline{V}}{(\mathtt{new}\ C(\overline{e})).f_i \to e_i}\ (\text{R-}\textsc{Field}) \qquad \frac{x : C \vdash x\ \mathtt{has}\ m(\overline{x} : \overline{T})\{c\} : T = e}{(\mathtt{new}\ C(\overline{e})).m(\overline{d}) \to e[\mathtt{new}\ C(\overline{e}), \overline{d}{=}\mathtt{this}, \overline{x}]}\ (\text{R-}\textsc{Invk})$$

$$\frac{e \to e'}{e.f_i \to e'.f_i}\ (\text{RC-}\textsc{Field}) \qquad \frac{e \to e'}{e.m(\overline{e}) \to e'.m(\overline{e})}\ (\text{RC-}\textsc{Invk-Recv})$$

$$\frac{\vdash C\{\mathtt{self} == \mathtt{new}\ C(\overline{d})\} <: T}{(T)(\mathtt{new}\ C(\overline{d})) \to \mathtt{new}\ C(\overline{d})}\ (\text{R-}\textsc{Cast}) \qquad \frac{e_i \to e_i'}{e.m(\ldots, e_i, \ldots) \to e.m(\ldots, e_i', \ldots)}\ (\text{RC-}\textsc{Invk-Arg})$$

$$\frac{e \to e'}{(T)e \to (T)e'}\ (\text{RC-}\textsc{Cast}) \qquad \frac{e_i \to e_i'}{\mathtt{new}\ C(\ldots, e_i, \ldots) \to \mathtt{new}\ C(\ldots, e_i', \ldots)}\ (\text{RC-}\textsc{New-Arg})$$

Figure 5. Semantics of FX

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

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

$$\frac{\mathtt{p}(\overline{\mathtt{T}}):\mathtt{o}\in\mathcal{C} \qquad \Gamma\vdash\overline{\mathtt{t}}:\overline{\mathtt{T}}}{\Gamma\vdash\mathtt{p}(\overline{\mathtt{T}}):\mathtt{o}} \quad (\textsc{Pred})$$

$$\frac{\mathtt{f}(\overline{\mathtt{T}}):\mathtt{T}\in\mathcal{C} \qquad \Gamma\vdash\overline{\mathtt{t}}:\overline{\mathtt{T}}}{\Gamma\vdash\mathtt{f}(\overline{\mathtt{T}}):\mathtt{T}} \quad (\textsc{Fun})$$

$$\frac{\Gamma\vdash\mathtt{t}_0:\mathtt{T}_0 \qquad \Gamma\vdash\mathtt{t}_1:\mathtt{T}_1 \qquad (\Gamma\vdash\mathtt{T}_0<:\mathtt{T}_1\vee\Gamma\vdash\mathtt{T}_1<:\mathtt{T}_0)}{\Gamma\vdash\mathtt{t}_0=\mathtt{t}_1:\mathtt{o}} \quad (\textsc{Equals})$$

No additional type judgements or transition rules are needed.

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* $\Gamma\vdash\mathtt{e}:\mathtt{T}$ *and* $\mathtt{e}\to\mathtt{e}'$ *then for some type* S, $\Gamma\vdash\mathtt{e}':\mathtt{S}$; $\mathtt{S}<:\mathtt{T}$.

The theorem needs the Substitution Lemma:

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

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

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

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

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

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

4. Discussion

todo: Move some of this to Section 6

4.1 Type properties versus type parameters

Type properties are similar, but not identical to type parameters. The differences may potentially confuse programmers used to Java generics or C++ templates. The key difference is that type properties are instance members and are thus accessible through access paths: `e.T` is a legal type.

Type properties, unlike type parameters, are inherited. For example, in the following code, `T` is defined in `List` and inherited into `Cons`. The property need not be declared by the `Cons` class.

```
class List[T] { }
class Cons extends List {
    def head(): T = { ... }
    def tail(): List[T] = { ... }
}
```

The analogous code for `Cons` using type parameters would be:

```
class Cons[T] extends List[T] {
    def head(): T = { ... }
    def tail(): List[T] = { ... }
}
```

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

4.2 Wildcards

Wildcards in Java [19, 51] were motivated by the following example (rewritten in X10 syntax) from [51]. Sometimes a class needs a field or method that is a list, but we don't care what the element type is. For methods, one can give the method a type parameter:

```
def aMethod[T](list: List[T]) = { ... }
```

This method can then be called on any `List` object. However, there is no way to do this for fields since they cannot be parametrized. Java introduced wildcards to allow such fields to be typed:

```
List<?> list;
```

In X10, a similar effect is achieved by not constraining the type property of `List`. One can write the following:

```
list: List;
```

Similarly, the method can be written without type parameters by not constraining `List`:

```
def aMethod(list: List) = { ... }
```

In X10, `List` is a supertype of `List[T]` for any `T`, just as in Java `List<?>` is a supertype of `List<T>` for any `T`. This follows directly from the definition of the type `List` as `List{true}`, and the type `List[T]` as `List{X==T}`, and the definition of subtyping.

Wildcards in Java can also be bounded. We achieve the same effect in X10 by using type constraints. For instance, the following Java declarations:

```
void aMethod(List<? extends Number> list) { ... }
<T extends Number> void aParameterizedMethod(List<T> list)
```

may be written as follows in X10:

```
def aMethod(list: List{T <: Number}) = { ... }
def aParameterizedMethod[T{self <: Number}](list: List[T])
```

Wildcard bounds may be covariant, as in the following example:

```
List<? extends Number> list = new ArrayList<Integer>();
Number num = list.get(0);     // legal
list.set(0, new Double(0.0)); // illegal
list.set(0, list.get(1));     // illegal
```

This can also be written in X10, but with an important difference:

```
list: List{T <: Number} = new ArrayList[Integer]();
num: Number = list.get(0);    // legal
list.set(0, new Double(0.0)); // illegal
list.set(0, list.get(1));     // legal! (when list is final)
```

Note because `list.get` has return type `list.T`, the last call in above is well-typed in X10; the analogous call in Java is not well-typed.

Finally, one can also specify lower bounds on types. These are useful for comparators:

```
class TreeSet[T] {
  def this[T](cmp: Comparator{T :> this.T}) { ... }
}
```

Here, the comparator for any supertype of `T` can be used as to compare `TreeSet` elements.

Another use of lower bounds is for list operations. The `map` method below takes a function that maps a supertype of the class parameter `T` to the method type parameter `S`:

```
class List[T] {
  def map[S](fun: Object{self :> T} => S) : List[S] = { ... }
}
```

4.3 Proper abstraction

Consider the following example adapted from [51]:

```
def shuffle[T](list: List[T]) = {
  for (i: int in [0..list.size()-1]) {
    val xi: T = list(i);
    val j: int = Math.random(list.size());
    list(i) = list(j);
    list(j) = xi;
  }
}
```

The method is parametrized on `T` because the method body needs the element type to declare the variable `xi`.

However, the method parameter can be omitted by using the type `list.T` for `xi`. Thus, the method can be declared with the signature:

```
def shuffle(list: List) { ... }
```

This is called *proper abstraction*.

This example illustrates a key difference between type properties and type parameters: A type property is a member of its class, whereas a type parameter is not. The names of type properties are visible outside the body of their class declaration.

In Java, Wildcard capture allows the 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 shuffle requires delegating to a private parametrized version that "opens up" the parameter.

4.4 Conditional methods and generlized constraints

For type parameters, method constraints are similar to generalized constraints proposed for C^J [12]. In the following code, the `T` parameter is covariant and so the `append` methods below are illegal:

```
class List[+T] {
  def append(other: T): List[T] = { ... }
      // illegal
  def append(other: List[T]): List[T] = { ... }
      // illegal
}
```

However, one can introduce a method parameter and then constrain the parameter from below by the class's parameter: For example, in the following code,

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

The constraints must be satisfied by the callers of `append`. For example, in the following code:

```
xs: List[Number];
ys: List[Integer];
xs = ys; // ok
xs.append(1.0); // legal
ys.append(1.0); // illegal
```

the call to `xs.append` is allowed and the result type is `List[Number]`, but the call to `ys.append` is not allowed because the caller cannot show that `Number<:Double`.

4.5 Self types

The generic constrained type system generalizes features of type systems from other object-oriented languages.

Type properties can also be used to support a form of self types [6, 7]. 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 [35] and J&'s dependent classes [34] 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 [6]. 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`.

4.6 Virtual types

Type properties share many similarities with virtual types [29, 28, 13, 14, 10] and similar constructs built on path-dependent types found in languages such as Scala [35], and J& [34]. Constrained types are more expressive than virtual types since they can be constrained at the use-site, can be refined on a per-object basis without explicit subclassing, and can be refined contravariantly as well as covariantly.

Thorup [49] proposed adding genericity to Java using virtual types. For example, a generic `List` class can be written as follows:

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

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

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

And this abstract class can be further refined to *final bind* `T` to a particular type:

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

These classes are related by subtyping: `IntList <: NumberList <: List`. Only classes where `T` is final bound can be non-abstract.

The analogous definition of `List` using type properties is as follows:

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

`NumberList` and `IntList` can be written as follows:

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

However, note that our version of `List` is not abstract. Instances of `List` can instantiate `T` with a particular type and there is no need to declared classes for `NumberList` and `IntList`. Instead, one can 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}`.

4.7 Structural constraints

Type constraints need not be limited to subtyping constraints. By introducing structural constraints on types, GFX allows type properties to be instantiated on any type with a given set of methods and fields. 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. The type system need not change except by extending the constraint system. The syntax for structural constraints is shown in Figure **??**. A structural constraint of the form `T has Sig` can specify that the type `T` have a constructor, method, or field of the given signature.

Structural constraints on types are found in many languages. Haskell supports type classes [26, 20]. 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,

```
constraints    c    ::=  ...
                         |  T has Sig
signatures     Sig  ::=  def this[X̄](x̄: T̄){c}: T
                         |  def m[X̄](x̄: T̄){c}: T
                         |  val x{c}: T
```

Figure 6. Grammar for structural constraints

and X10. The language PolyJ [33] 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 is:

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

A structural constraint is satisfied if the type has a member of the appropriate name and with a compatible type. The constraint `Z has def m[X̄](x̄:T̄): U` is satisfied by a type `T` if it has a method `m` with signature `def m[Ȳ](ȳ:S̄): V` and where `([Ȳ](ȳ:S̄) => V)[T=Z]` is a subtype of `([X̄](x̄:T̄) => U)[T=Z]`. As an example, the constraint `X has def compareTo(X): int` is satisfied by both of the following classes:

```
class C { def compareTo(x: C): int = ...; }
class D { def compareTo(x: Object): int = ...; }
```

todo: The important bit about all this is... don't have to change the type system, just the constraints

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 [9, 2, 3] and the implementation of PolyJ [33]. 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.

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

```
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 [32] and by Reynolds [41]. These and subsequent systems are based on constraints over types, but not over values. Trifonov and Smith [52] proposed a type system in which types are refined using subtyping constraints. Pottier [37, 39] presents a constraint-based type system for an ML-like language with subtyping.

These developments lead to HM(X) [46, 38, 40], a constraint-based framework for Hindley–Milner-style type systems. The framework is parameterized 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 [47] showed that the type inference algorithm for HM(X) can be encoded as a constraint logic program parameterized 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 [20]. PolyJ [33] also uses structual constraints to bound type parameters.

Dependent types. Dependent type systems [55, **?**, 30, 4] parametrize types on values. Refinement type systems [18, 1, 27, 21, 15, 16, 42], introduced by Freeman and Pfenning [18], 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, [55], 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 ML program. This differs from generic constrained types, where erasure of subtyping constraints can prevent the program from type-checking. DML does not permit any runtime 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, specifically ML, 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

```
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

field selection and equality on object references. Note that DML-style constraints are easily encoded in constrained types.

Logically qualified types, or liquid types [42], 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 overapproximated 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 [15, 16] 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 GFX, 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 [17] extends types in OCaml [**?**] with constraints written as Coq [11] 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 impedence mismatch caused by the differing syntax, representation, and behavior of OCaml versus Coq.

Genericity. Genericity in object-oriented languages is usually suported through type parameterization.

A number of proposals for adding genericity to Java quickly followed the initial release of the language [5, 36, 33, 49, 2]. GJ [5] implements invariant type parameters via type erasure. PolyJ [33] supports run-time representation of types via adapter objects, and also permits instantiation of parameters on primitive types and structural parameter bounds. LM [54, 53] also supports a run-time representation of types. NextGen [9, 2] was implemented using run-time instantiation. X10's generics have a hybrid implementation, adopting PolyJ's adapter object approach for dependent types and for type introspection and using NextGen's run-time instantiation approach for greater efficiency.

C^{J} also supports generics via [48]. Type parameters may be declared with definition-site variance tags. Compilation does not erase types; the CLR performs run-time parameter instantiation. Generalized type constraints were proposed for C^{J} [12]. Methods can be annotated with subtyping constraints that must be satisfied to invoke the method. Generic X10 supports these constraints, as well as constraints on values, with method and constructor where clauses.

GFX does not support *bivariance* [24]; 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 GFX simply by leaving `X` unconstrained.

Virtual classes and types. *Virtual classes* [28, 29, 14]. are a language-based extensibility mechanism that where originally introduced in the language BETA [28] as a mechanism for supporting genericity. Virtual classes in BETA are not statically type safe, but this has been remedied in recent formulations [13, 14] and in variants of virtual classes [35, 34, 10, 25] using path-dependent types.

Virtual types, also introduced in BETA [28], 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 [49] proposed extending Java with virtual types as a genericity mechanism. Virtual types influenced Java's wildcards [51, 19, 8].

Igarashi and Pierce [23] 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 [50] and extended for parameterized type systems by Igarashi and Viroli [24]. The latter type system lead to the development of wildcards in Java [19, 51, 8].

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^{J} 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] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

[13] 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.

[14] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. pages 270–282, Charleston, South Carolina, January 2006.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell.

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

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

[43] 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.

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

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

[46] 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.

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

[48] 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.

[49] Kresten Krab Thorup. Genericity in Java with virtual types. Number 1241 in Lecture Notes in Computer Science, pages 444–471. Springer-Verlag, 1997.

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

[51] 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.

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

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

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

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