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 (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 $\mathcal{C}$ 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

todo: Awkward, repetitive

todo: More positioning, relative to: DML, HM(X), constrained types (Trifonov, Smith) and subtyping constraints, Java generics, GJ, PolyJ, C# 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 strong program invariants as types that are checked by the compiler, without sacrificing the ability to reuse code.

Examples include generics, self types, dependent types. 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 extensions of the core dependent type system: generic types, virtual types, and self types among them.

We have formalized the type system in an extension of Featherweight Java [19] and provide proof of soundness. The type system is parametrized on the constraint system. By augmenting the default constraint system, the type system can serve as a core calculus for formalizing extensions of a core object-oriented language.

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.

This work is done in the context of the X10 programming language [35]. 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 [35, 34], 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 [18] and Scala [27] and in functional languages such as ML [24] and Haskell [20].

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,

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

`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. The `Cell` defined class above is a generic class. X10 supports equality constraints, written $\mathtt{T}_1\mathtt{==T}_2$, and subtyping constraints, written $\mathtt{T}_1\mathtt{<:T}_2$, 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 **??**.

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 [27] and J&'s dependent classes [26] 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.

Virtual types. Type properties share many similarities with virtual types [22, 21, 14, 15, 10] and similar constructs built on path-dependent types found in languages such as Scala [27], and J& [26]. 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. We explore this connection in Section 2.3.

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 [18, 39, 7, 29, 6, 37, 1, 2, 12, 13, 27], 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.

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.

Outline. The rest of the paper is organized as follows. Section 2 discusses related work. An informal overview of generic constrained types in X10 is presented in Section 3. Section 6 presents a formal semantics and a proof of soundness. The implementation of generics in X10 by translation to Java is described in Section 8. Finally, Section 10 concludes.

todo: Fix this

2. Related work

2.1 Dependent types

[23, 41, 28, 4, 5, 3, 11]

Liquid types [33]

Hybrid types [16, 17]

Constrained types [36] [40] [32]

Subtyping constraints [30]

HM(X) [31]

2.2 Generics

[38] [39] [13] [25] [6] [2] [1]

2.3 Virtual types

Thorup [37] 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}.

3. X10 language overview

This section presents an informal description of the generic constrained types in X10. The type system is formalized in a simplified version of X10, GFX (Generic Featherweight X10), in Section 6.

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.

3.1 Classes

Classes in X10 may be declared with any number of type properties and value properties. These properties can be constrained with a *class invariant*, a predicate on the properties of any instance of the class. The general form of a class declaration is:

$$\texttt{class C[}\overline{\texttt{X}}\texttt{](}\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{)\{c\} extends D\{d\}}$$
$$\texttt{implements }\overline{\texttt{I}}\texttt{\{}\overline{\texttt{c}}\texttt{\} \{ ... \}}$$

This declaration defines a class C with zero or more type properties $\overline{\texttt{X}}$, zero of more value properties $\overline{\texttt{x}}$ of types $\overline{\texttt{T}}$, and a class invariant c. The class C is a subclass of D (constrained by d) and implements the constrained interfaces $\overline{\texttt{I}}\{\overline{\texttt{c}}\}$.

Both classes and interfaces may define properties. Value properties may be considered to be public final instance fields. Whereas Java supports only static fields in interfaces, X10 allows interfaces to define value properties. Any class implementing an interface must declare or inherit from a superclass the properties inherited from the interface. All properties of a class, both type and value, must be initialized by the class's constructors.

Classes may define fields, methods, and constructors. Consider the example in Figure 1. The declaration syntax is similar to Scala's. Fields may be declared either val or var. A val field is *final* and must be assigned exactly once by the constructor. Methods are declared with a def keyword. Methods in classes and interfaces may be declared static. Mutable static fields are not permitted. Constructor syntax is similar to method syntax; X10 adopts Scala's syntax, using the name this for constructors. In X10, constructors have a return type, which constrains the properties of the new object.

3.2 Constrained types

A constrained type 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.

For brevity, the constraint may be omitted and interpreted as true. The syntax $\texttt{C[T}_1\texttt{,}\ldots\texttt{,T}_m\texttt{](e}_1\texttt{,}\ldots\texttt{,e}_n\texttt{)}$ is sugar for $\texttt{C\{X}_1\texttt{==T}_1\texttt{,}\ldots\texttt{,X}_m\texttt{==T}_m\texttt{,x}_1\texttt{==e}_1\texttt{,}\ldots\texttt{,x}_n\texttt{==e}_n\texttt{\}}$ 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.

In this shortened syntax, a type argument T used may also be annotated with a *use-site variance tag*, either + or -: if X is a type property, then the syntax C[+T] is sugar C{X<:T} and C[-T] is sugar C{X:>T}; of course, C[T] is sugar C{X==T}. Use-site variance is discussed in more detail in Section 3.4

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 equality and subtyping consraints. Compiler plugins may be installed to handle richer constraint systems such as Presburger arithmetic or set constraints.

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

    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;
    }

    def map[S](f: (T)=>S): List[S] = ...;

    // XXX
    def get(i: int){0 <= i, i < length} = {
        if (i == 0) return head;
        if (tail != null) return tail.get(i-1);
        throw new IndexOutOfBoundsException();
    }
}
```

Figure 1. List example

3.3 Generics

Type properties and subtyping constraints are used in X10 to provide genericity.

Unlike existing existing proposals for generic types in Java-like languages [18, 39, 7, 29, 6, 37, 1, 2, 12, 13, 27], 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.

3.4 Type constraints and variance

Type properties and subtyping constraints are used in X10 to provide genericity.

The List class in Figure 1. Consider the following subtypes of List.

- 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. In the following code, y cannot be passed to the set method because it is not known if Object is a subtype of c.T.

```
val x: l.T = l.get(0);
val y: Object = l.get(1);
l.set(x); // legal
l.set(y); // illegal
```

- List{T==float}. The type property T is bound to float. Assuming l has this type, then following code is legal:

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

The type of `l.get()` is `l.T`, which is equivalent to `float`.

- `List{T<:Collection}`. This type constrains `T` to be a subtype of `Collection`. All instances of this type must bind `T` to a subtype of `Collection`; for example `List[Set]` (i.e., `List{T==Set}` is a subtype of `List{T<:Collection}` because `T==Set` entails `T<:Collection`. If `l` has the type `List{T<:Collection}`, then `l.get(0)` 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. The `set` method may be called with any supertype of `String`; the return type of the `get` method is known to be a supertype of `String` (and implicitly a subtype of `Object`).

3.5 Type properties

Type properties may be declared invariant, covariant, or contravariant. If a property `X` of a class `C` is covariant, then if `S` is a subtype of `T`, the type `C{X==S}` is a subtype of `C{X==T}`. Similarly, if `X` is contravariant, `C{X==T}` is a subtype of `C{X==S}`. It is illegal for a covariant property to occur in a negative position in its class declaration and for a contravariant property to occur in a positive position. A position is negative if it is a formal parameter type, or occurs in a method where clause. A position is positive if it is a return type or occurs in a method where clause.

3.6 Constructors

Objects in X10 are initialized with constructors. Constructors are defined using the syntax `def this`, illustrated with the three `List` constructors in Figure 1.

Constructors 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.

Constructors can take zero or more type parameters and zero or more value parameters.

Properties are initialized with a `property` statement. The `property` statement is used to set all the properties of the new object simultaneously; the syntax is similar to a `super` constructor call. The first constructor takes zero arguments and initializes the type property `T` to the type parameter `S` length to `0`. The second constructor initializes the length to `1`, the third to one plus the length of the tail.

Constructors have "return types" that can specify an invariant satisfied by the object being constructed. This type is used as the type of the `new` expression that invoked the constructor. The compiler verifies that the constructor return type and the class invariant are implied by the `property` statement and any `super` or `this` calls in the constructor body.

Classes that do not declare a constructor have a default constructor with a type parameter for each type property and a value parameter for each value property.

3.7 Methods

Methods in X10 are declared with the `def` keyword. The `List` class in Figure 1 declares methods `get` and `map`.

Like Java, X10 supports both instance and static methods. Since a type property is an instance member, a static method may not refer to a type property of the class.

Interfaces are also permitted to have static methods. Classes implementing the interface must provide an implementation of the static methods of the interface. This feature is useful when a type property `T` is constrained to implement an interface `I`; static methods of `I` can be invoked through `T`.

Methods may have both type and value parameters. For instance, the `map` method in Figure 1 has a type parameter `S` and a value parameter that is a function from `T` to `S`.

A parametrized method can is invoked by giving type arguments before the expression arguments. The following code takes a list of `String`s and returns a list of string lengths of type `int`

```
xs: List[String] = ...;
ys: List[int] = xs.map[int](
        (x: String) => x.length());
```

Conditional methods. Methods and constructor may also have *where clauses*, constraints on how the method may be invoked. The where clause is written after the method parameters and before the return type. The `get` method in Figure 1 requires that the argument `i` is within the list bounds. A method with a where clause is called a *conditional method*.

For type parameters, method where clauses are similar to generalized constraints proposed for C# [13]. 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`.

Method overriding. 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 where clause may be weakened by an overriding method; that is, the where clause of the superclass must entail the where clause of the subclass.

3.8 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 2. 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

```
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 2. A `SortedList` class with function-typed value properties

```
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 3. Grammar for structural constraints

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.

4. Structural constraints

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 11. 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 [20] supports type classes. 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 [6] allows type parameters to be bounded using structural where clauses. For example, a sorted list class in PolyJ can be written as follows:

```
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): void = { ... x.compareTo(y) ... }
}
```

A constraint is satisfied if the type has a member of the appropriate name and with a compatible type. The constraint `X has def` $\mathtt{m(T_1): T_2}$ is satisfied by a type `T` if it has a method `m` whose type is a subtype of $(\mathtt{T_1} \Rightarrow \mathtt{T_2})[\mathtt{T}/\mathtt{X}]$. As an example, the constraint `X has def equals(X): boolean` is satisfied by all three of the following classes:

```
class C { def equals(x: C): boolean; }
class D extends C { }
class E { def equals(x: Object): boolean; }
```

By using function types and where clauses on constructors, X10 can go further than PolyJ. Unlike in PolyJ, where the `compare` method must be provided by `T`, in X10 the `compare` function can be external to `T`. This is achieved as follows:

```
class SortedList[T] {
    val compare: (T,T) => int;
    def this(cmp: (T,T) => int) = { compare = cmp; }
    def add(x: T) = { ... compare(x,y) ... }
}
```

This permits `SortedList` to be instantiated using different compare functions:

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

But, a problem with this approach is that the compare function must be provided to the constructor at each instantiation of `SortedList`. The problem can be resolved by using constructors with different structural constraints:

```
class SortedList[T] {
    val compare: (T,T) => int;
    def this[T]() where T has compareTo(T): int = {
        this[T](T.compareTo.(S));
    }
    def this[T](cmp: (T,T) => int) = { compare = cmp; }
    def add(x: T) = { ... compare(x,y) ... }
}
```

Now, `SortedList` can be instantiated with any type that has a `compareTo` method without explicitly specifying the method at each constructor call.

5. Self types

X10 itself does not support self types [8, 9] directly, but type properties can be used to encode them.

We introduce a type property `class` to the root of the class hierarchy, `Object`:

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

Scala's path-dependent types [27] and J&'s dependent classes [26] take a similar approach.
Self types are achieved by implicitly constraining types so that if a 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;
    }
}
```

The implementation of the method is free to access the `bits` field of the argument since the constraint `this.class<:BitSet` ensures the field is accessible.

6. Formal semantics

We present a core calculus, GFX, for X10 with generics. GFX is based on Constrained Featherweight Java [34].

todo: Add method overriding rules: covariant return, contravariant args, weaker constraints

The grammar for GFX is shown in Figure 4. The calculus elides features of the full X10 language not relevant to this paper.

Figure 5 extends the grammar with syntactic sugar for subtyping constraints and existential types. The subtyping constraint $\mathtt{t}_1 \mathrel{<:} \mathtt{t}_2$ is atomic formula. The existential type $\exists \mathtt{x}:\mathtt{T}.\ \mathtt{R\{c\}}$ is sugar for $\mathtt{R}\{\exists \mathtt{x}.\ \sigma(\mathtt{x}:\mathtt{T}), \mathtt{c}\}$.

We assume a fixed but unknown constraint system $\mathcal{D}$. A program P is written using constraints from $\mathcal{D}$, We assume classes defined in P do not have a cyclic inheritance structure.

$$\frac{\mathtt{extends}^{+}\ \text{acyclic}}{\vdash \overline{\mathtt{L}}\ \text{ok}} \qquad (\text{PROGRAM OK})$$

6.1 The object constraint system, $\mathcal{O}$

From P and $\mathcal{D}$ we generate an *object constraint system* $\mathcal{O}$, shown in Figure 6, as follows. Let C and D range over names of classes in P, f over field names, m over method names, T over types, and c over constraints in the underlying data constraint system $\mathcal{D}$.

In the method signature $[\overline{\mathtt{X}}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}$, the type variables $\overline{\mathtt{X}}$ and data variables $\overline{\mathtt{x}}$ are considered bound; formulas with bound variables are considered equivalent up to α-renaming.

The constraint system satisfies the axioms and inference rules in Figure 6. The `class`, `extends`, `fields`, and `mtype` constraints are given directly from the program P.

The constraint system $\mathcal{C}$ is the disjoint conjunction $\mathcal{D}$, $\mathcal{O}$ of the constraint systems $\mathcal{D}$ and $\mathcal{O}$. (This requires the assumption that $\mathcal{D}$ does not have any constraints in common with $\mathcal{O}$.

6.2 Structural and logical rules

All judgments are intuitionistic. In particular, this means that all constraint systems satisfy the rules and axioms in Figure 7.

6.3 Well-formedness rules

We use the judgment for well-typedness for expressions to represent well-typedness for constraints. That is, we posit a special type o (traditionally the type of propositions), and regard constraints as expressions of type o.

program	P	::=	$\overline{\mathtt{L}}$
classes	L	::=	$\mathtt{class\ C}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}$ $\mathtt{extends\ T}\ \{\ \overline{\mathtt{M}}\ \}$
base types	R		
classes		::=	C
type variables		|	X
type members		|	e.X
type type		|	type
types	T	::=	R{c}
methods	M	::=	$\mathtt{def\ m}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\ \overline{\mathtt{T}})\{\mathtt{c}\}\mathtt{:\ T = e}$
expressions	e		
literals		::=	true | false | null | n
variables		|	x
field access		|	e.x
call		|	$\mathtt{e}_0\mathtt{.m}[\overline{\mathtt{T}}](\overline{\mathtt{e}})$
new		|	$\mathtt{new\ C}[\overline{\mathtt{T}}](\overline{\mathtt{e}})$
cast		|	e as T
constraint terms	t		
self		::=	self
variables		|	x
properties		|	t.x
atoms		|	$\mathtt{g}(\mathtt{t}_1,\ldots,\mathtt{t}_n)$
new		|	$\mathtt{new\ C}(\mathtt{t}_1,\ldots,\mathtt{t}_n)$
constraint	c		
true		::=	true
equality		|	$\mathtt{t}_1 \mathtt{==} \mathtt{t}_2$
existentials		|	$\exists \mathtt{x}.\ \mathtt{c}$
conjunction		|	$\overline{\mathtt{c}}$
predicates		|	$\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_n)$
environments	Γ	::=	ε
		|	$\Gamma, \mathtt{c}$
		|	$\Gamma, \mathtt{x}:\mathtt{T}$
		|	$\Gamma, \mathtt{X}:\mathtt{type}$

Figure 4. GFX grammar

types	T	::=	...
		|	$\exists \mathtt{x}:\mathtt{T}_0.\ \mathtt{T}$
		|	$\exists \mathtt{X}:\mathtt{type}.\ \mathtt{T}$
constraint terms	t	::=	...
literals			true | n | C
type variables		|	X
type properties		|	t.X
constraint	c	::=	...
		|	$\mathtt{t}_1 \mathrel{<:} \mathtt{t}_2$
		|	cons(T,z)

Figure 5. GFX grammar with subtyping constraints

Further, we change the formulation slightly so that there are no constraints of the form $\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_n)$; rather instance method invocation syntax is used to express invocation of pre-defined constraints. This logically leads to the step of simply marking certain classes as "predicate" classes—all the (instance) methods of these classes whose return type is o then correspond to "primitive constraints". Syntactically, we continue to use the symmetric syntax $\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_n)$ rather than $\mathtt{t}_1\mathtt{.p}(\mathtt{t}_2,\ldots,\mathtt{t}_n)$. The alternative is tor introduce static methods and static method invocations in the expression language. This is not difficult, but is annoying to have to repeat most of the formulation of instance methods.

This means that the only cases left to handle are all the simple ones, expression the availability of certain constraints and operations of type o.

$$
\begin{array}{llll}
\text{constraint} & \mathtt{c} & ::= & \mathtt{class(C)} \\
& & | & \mathtt{C\ extends\ D} \\
& & | & \mathtt{fields(x,\overline{f}:\overline{T})} \\
& & | & \mathtt{mtype(x,m,[\overline{X}(\overline{x}:\overline{T})\{c\}}
\end{array}
$$

$$
\frac{\mathtt{class\ C[\overline{X}](\overline{f}:\overline{T})\{c\}\ extends\ D\{d\}\ \{\overline{M}\}} \in \mathtt{P}}{\begin{array}{c}\vdash_O \mathtt{class(C)} \\ \vdash_O \mathtt{C\ extends\ C} \\ \vdash_O \mathtt{C\ extends\ D}\end{array}}
$$

$$
\frac{\mathtt{class\ C[\overline{X}](\overline{f}:\overline{T})\{c\}\ extends\ Object\ \{\overline{M}\}} \in \mathtt{P}}{\Gamma, \mathtt{z}:\mathtt{C}\{d\} \vdash_O \mathtt{fields(z,\overline{f}:\overline{T})}} \quad (\text{FIELDS})
$$

$$
\frac{\begin{array}{c}\vdash_O \mathtt{C\ extends\ D} \\ \mathtt{class\ C[\overline{X}](\overline{f}:\overline{T})\{c\}\ extends\ D\{d\}\ \{\overline{M}\}} \in \mathtt{P} \\ \Gamma, \mathtt{z}:\mathtt{D\{d\}} \vdash_O \mathtt{fields(z,\overline{f}_0:\overline{T}_0)}\end{array}}{\Gamma, \mathtt{z}:\mathtt{D\{d\}} \vdash_O \mathtt{fields(z,\overline{f}_0:\overline{T}_0,\overline{f}:\overline{T})}} \quad (\text{FIELDS-EXTENDS})
$$

$$
\frac{\begin{array}{c}\mathtt{class\ C[\overline{X}](\overline{f}:\overline{T})\{c\}\ extends\ Object\ \{\overline{M}\}} \in \mathtt{P} \\ \mathtt{M}_i = \mathtt{def\ m}_i\mathtt{[\overline{X}](\overline{x}:\overline{T})\{c\}:\ T\ =\ e}\end{array}}{\Gamma, \mathtt{z}:\mathtt{C\{d\}} \vdash_O \mathtt{mtype(z,m}_i\mathtt{,[\overline{X}](\overline{x}:\overline{T})\{c\}} \rightarrow \mathtt{T)}} \quad (\text{MTYPE})
$$

Figure 6. The constraint system O

$$
\Gamma, \mathtt{c} \vdash \mathtt{c} \quad (\text{ID})
$$

$$
\frac{\Gamma \vdash \mathtt{c} \qquad \Gamma, \mathtt{c} \vdash \mathtt{d}}{\Gamma \vdash \mathtt{d}} \quad (\text{CUT})
$$

$$
\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \mathtt{T}:\mathtt{type} \qquad \mathtt{x} \notin \mathit{var}(\Gamma)}{\Gamma, \mathtt{x}:\mathtt{T} \vdash \phi} \quad (\text{WEAK-1})
$$

$$
\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \mathtt{c}:\mathtt{o}}{\Gamma, \mathtt{c}:\phi} \quad (\text{WEAK-2})
$$

$$
\frac{\Gamma, \psi_0, \psi_1 \vdash \phi}{\Gamma, (\psi_0, \psi_1) \vdash \phi} \quad (\text{AND-L})
$$

$$
\frac{\Gamma \vdash \psi_0 \qquad \Gamma \vdash \psi_1}{\Gamma \vdash (\psi_0, \psi_1)} \quad (\text{AND-R})
$$

$$
\frac{\Gamma \vdash \phi[\mathtt{t}/\mathtt{x}]}{\Gamma \vdash_C \exists \mathtt{x}.\ \phi} \quad (\text{EXISTS-R})
$$

$$
\frac{\Gamma, \mathtt{x}:\mathtt{T}, \psi \vdash \phi \qquad \mathtt{x}\ \text{fresh}}{\Gamma, \exists \mathtt{x}:\mathtt{T}.\ \psi \vdash \phi} \quad (\text{EXISTS-L})
$$

Figure 7. Logical rules

$$
\Gamma \vdash \mathtt{true}:\mathtt{o} \quad (\text{TRUE})
$$

$$
\frac{\Gamma \vdash \mathtt{t}_0:\mathtt{T}_0 \qquad \Gamma \vdash \mathtt{t}_1:\mathtt{T}_1 \qquad (\Gamma \vdash \mathtt{T}_0 <: \mathtt{T}_1 \vee \Gamma \vdash \mathtt{T}_1 <: \mathtt{T}_2)}{\Gamma \vdash \mathtt{t}_0 == \mathtt{t}_1 : \mathtt{o}} \quad (\text{EQUALS})
$$

$$
\frac{\Gamma \vdash \mathtt{c}_0:\mathtt{o} \qquad \Gamma \vdash \mathtt{c}_1:\mathtt{o}}{\Gamma \vdash (\mathtt{c}_0, \mathtt{c}_1):\mathtt{o}} \quad (\text{AND})
$$

$$
\frac{\Gamma \vdash \mathtt{t}:\mathtt{T} \qquad \Gamma \vdash \mathtt{c}[\mathtt{t}/\mathtt{x}]:\mathtt{o}}{\Gamma \vdash \exists \mathtt{x}:\mathtt{T}.\ \mathtt{c}:\mathtt{o}} \quad (\text{SOME})
$$

$$
\frac{\Gamma \vdash \mathtt{class(C)} \qquad \Gamma, \mathtt{self}:\mathtt{C} \vdash \mathtt{c}:\mathtt{o}}{\Gamma \vdash \mathtt{C\{c\}}:\mathtt{type}} \quad (\text{TYPE})
$$

Figure 8. Well-formedness rules

6.4 Subtyping constraints

6.5 Constraint projection

First, for a type environment Γ, we define the *constraint projection*, $\sigma(\Gamma)$ thus:

$$
\begin{aligned}
\sigma(\varepsilon) &= \mathtt{true} \\
\sigma(\Gamma, \mathtt{x}:\mathtt{T}) &= \sigma(\Gamma), \mathit{cons}(\mathtt{T}, \mathtt{x}) \\
\sigma(\Gamma, \mathtt{c}) &= \sigma(\Gamma), \mathtt{c}
\end{aligned}
$$

The auxiliary function *cons* specifies the constraint for a type `T` with `self` bounds to `x`. The constraint projection uses an atomic formula `cons`, which is equated to the constraint of `T` if `T` is not a type variable.

$$
\begin{aligned}
\mathit{cons}(\mathtt{C}, \mathtt{z}) &= \mathtt{cons(C,z)} \\
\mathit{cons}(\mathtt{C\{c\}}, \mathtt{z}) &= \mathtt{c}[\mathtt{z}/\mathtt{self}], \mathtt{cons(C\{c\},z)} \mathtt{==} \mathtt{c}[\mathtt{z}/\mathtt{self}] \\
\mathit{cons}(\mathtt{p.X}, \mathtt{z}) &= \mathtt{cons(p.X,z)} \\
\mathit{cons}(\mathtt{X}, \mathtt{z}) &= \mathtt{cons(X,z)}
\end{aligned}
$$

Thus, for example, the constraint projection of the environment:

```
b: D, a: C{self.X==D{d},self.Y<:b.Z}
```

is:

```
a.X==D{d}, a.Y<:b.Z
```

6.6 Type well-formedness

6.7 Type inference rules

6.7.1 Constraint rules

6.7.2 Expression typing judgment

The cast rule T-CAST requires that the cast type be well-formed.

The field access rule T-FIELD differs from the rule in the paper in that there is no need to substitute a fresh variable for the receiver. Note that `this` may be free in `S`—that would be a reference to the current object in the code in which `e.f` occurs, not a reference to the receiver of the `e.f` field selection (i.e., the object obtained by evaluating `e`).
if we allow adding constraints to arbitrary types—do we?

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{==} \mathtt{T}_2}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{cons(T_1,z)} \mathtt{==} \mathtt{cons(T_2,z)}} \quad (\textsc{Cons-eq})$$

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{<:} \mathtt{T}_2}{\overline{\mathtt{c}}, \mathtt{cons(T_1,z)} \vdash_{\mathcal{C}} \mathtt{cons(T_2,z)}} \quad (\textsc{Cons-sub})$$

$$\vdash_{\mathcal{C}} \mathtt{cons(C,z)} \quad (\textsc{Cons})$$

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \overline{\mathtt{s}} \mathtt{==} \overline{\mathtt{t}}}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{f}(\overline{\mathtt{s}}) \mathtt{==} \mathtt{f}(\overline{\mathtt{t}})} \quad (\textsc{Eq-atom})$$

$$\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{t} \mathtt{==} \mathtt{t} \quad (\textsc{Eq-refl})$$

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{t}_1 \mathtt{==} \mathtt{t}_2 \qquad \overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{t}_2 \mathtt{==} \mathtt{t}_3}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{t}_1 \mathtt{==} \mathtt{t}_3} \quad (\textsc{Eq-trans})$$

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{t}_1 \mathtt{==} \mathtt{t}_2}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{t}_2 \mathtt{==} \mathtt{t}_1} \quad (\textsc{Eq-sym})$$

$$\frac{\begin{array}{c}\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{<:} \mathtt{T}_2 \\ \overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_2 \mathtt{<:} \mathtt{T}_1\end{array}}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{==} \mathtt{T}_2} \quad (\textsc{Eq-sub})$$

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{C\{c\}} : \mathtt{type} \qquad \overline{\mathtt{c}}, \mathtt{c} \vdash_{\mathcal{C}} \mathtt{d}}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{C\{c\}} \mathtt{<:} \mathtt{C\{d\}}} \quad (\textsc{Sub-cons})$$

$$\frac{\mathtt{C}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{T}})\mathtt{\{c\}\ ext\ T\ \{\ K}\ \overline{\mathtt{M}}\ \overline{\mathtt{F}}\ \mathtt{\}}}{\vdash_{\mathcal{C}} \mathtt{C} \mathtt{<:} \mathtt{T}} \quad (\textsc{Sub-super})$$

$$\vdash_{\mathcal{C}} \mathtt{T} \mathtt{<:} \mathtt{Object} \quad (\textsc{Sub-object})$$

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{==} \mathtt{T}_2}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{<:} \mathtt{T}_2} \quad (\textsc{Sub-eq})$$

$$\frac{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{<:} \mathtt{T}_2 \qquad \overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_2 \mathtt{<:} \mathtt{T}_3}{\overline{\mathtt{c}} \vdash_{\mathcal{C}} \mathtt{T}_1 \mathtt{<:} \mathtt{T}_3} \quad (\textsc{Sub-trans})$$

Figure 9. Equality and subtyping rules

$$\frac{\mathtt{C}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{T}})\mathtt{\{c\}\ ext\ T\ \{}\ \overline{\mathtt{M}}\ \overline{\mathtt{F}}\ \mathtt{\}}}{\vdash \mathtt{C} : \mathtt{type}}$$

$$\frac{\Gamma \vdash \mathtt{T} : \mathtt{type} \qquad \Gamma, \mathtt{self} : \mathtt{T} \vdash \mathtt{c} : \mathtt{Boolean} \qquad \sigma(\Gamma) \vdash_{\mathcal{C}} \mathtt{c}\ \mathrm{OK}}{\Gamma \vdash \mathtt{T\{c\}} : \mathtt{type}}$$

$$\frac{\Gamma \vdash \mathtt{p} : \mathtt{T} \qquad \Gamma \vdash \mathtt{T}\ \mathrm{has}\ \mathtt{X}}{\Gamma \vdash \mathtt{p.X} : \mathtt{type}}$$

$$\Gamma, \mathtt{X} : \mathtt{type} \vdash \mathtt{X} : \mathtt{type}$$

Figure 10. Type well-formedness

$$\frac{\mathtt{C}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{T}})\mathtt{\{c\}\ ext\ T\ \{\ K}\ \overline{\mathtt{M}}\ \overline{\mathtt{F}}\ \mathtt{\}}}{\begin{array}{c}\vdash \mathtt{C}\ \mathrm{has}\ \mathtt{K} \\ \vdash \mathtt{C}\ \mathrm{has}\ \mathtt{X}_i \\ \vdash \mathtt{C}\ \mathrm{has}\ \mathtt{x}_i : \mathtt{T}_i \\ \vdash \mathtt{C}\ \mathrm{has}\ \mathtt{M}_i \\ \vdash \mathtt{C}\ \mathrm{has}\ \mathtt{F}_i\end{array}} \quad (\textsc{Has-class})$$

$$\frac{\mathtt{Z} \neq \mathtt{K} \qquad \Gamma \vdash \mathtt{T}_1\ \mathrm{has}\ \mathtt{Z} \qquad \sigma(\Gamma) \vdash \mathtt{T}_2 \mathtt{<:} \mathtt{T}_1}{\Gamma \vdash \mathtt{T}_2\ \mathrm{has}\ \mathtt{Z}} \quad (\textsc{Has-sub})$$

Figure 11. Structural constraints

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \sigma(\Gamma) \vdash_{\mathcal{C}} \mathtt{S} \mathtt{<:} \mathtt{T} \qquad \Gamma \vdash \mathtt{T} : \mathtt{type}}{\Gamma \vdash \mathtt{e} : \mathtt{T}} \quad (\textsc{T-sub})$$

$$\begin{array}{c}\vdash \mathtt{true} : \mathtt{Boolean\{self==true\}} \\ \vdash \mathtt{false} : \mathtt{Boolean\{self==false\}}\end{array} \quad (\textsc{T-bool})$$

$$\vdash n : \mathtt{Int\{self==}n\mathtt{\}} \quad (\textsc{T-int})$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{e}_1 \mathtt{==} \mathtt{e}_2 : \exists \mathtt{z}_1 : \mathtt{T}_1, \mathtt{z}_2 : \mathtt{T}_2.\ \mathtt{Boolean\{self==(z_1==z_2)\}}} \quad (\textsc{T-eq})$$

$$\frac{\Gamma \vdash \mathtt{T}_1 : \mathtt{type} \qquad \Gamma \vdash \mathtt{T}_2 : \mathtt{type}}{\Gamma \vdash \mathtt{T}_1 \mathtt{==} \mathtt{T}_2 : \mathtt{Boolean}} \quad (\textsc{T-teq})$$

$$\frac{\Gamma \vdash \mathtt{T}_1 : \mathtt{type} \qquad \Gamma \vdash \mathtt{T}_2 : \mathtt{type}}{\Gamma \vdash \mathtt{T}_1 \mathtt{<:} \mathtt{T}_2 : \mathtt{Boolean}} \quad (\textsc{T-tsub})$$

$$\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T} \quad (\textsc{T-var})$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \Gamma \vdash \mathtt{T} : \mathtt{type}}{\Gamma \vdash \mathtt{e\ as\ T} : \mathtt{T}} \quad (\textsc{T-cast})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{T} \\ \mathtt{T}\ \mathrm{has}\ \mathtt{f\{c\}} : \mathtt{U} \\ \sigma(\Gamma, \mathtt{this} : \mathtt{T}) \vdash_{\mathcal{C}} \mathtt{c}\end{array}}{\Gamma \vdash \mathtt{e.f} : \exists \mathtt{this} : \mathtt{T}.\ \mathtt{U\{self==this.f\}}} \quad (\textsc{T-field})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \\ \mathtt{T}_0\ \mathrm{has}\ \mathrm{def}\ \mathtt{m}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{S}})\mathtt{\{c\}} : \mathtt{U} = \mathtt{e} \\ \Gamma' = \Gamma, \overline{\mathtt{X}} : \mathtt{type}, \mathtt{this} : \mathtt{T}_0, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \overline{\mathtt{X}} \mathtt{==} \overline{\mathtt{V}} \\ \sigma(\Gamma') \vdash_{\mathcal{C}} \mathtt{c} \\ \sigma(\Gamma') \vdash_{\mathcal{C}} \overline{\mathtt{T}} \mathtt{<:} \overline{\mathtt{S}}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}[\overline{\mathtt{V}}](\overline{\mathtt{e}}) : \exists \overline{\mathtt{X}} : \mathtt{type}, \mathtt{this} : \mathtt{T}_0, \overline{\mathtt{x}} : \overline{\mathtt{T}}.\ \mathtt{U}} \quad (\textsc{T-invk})$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \\ \mathtt{C}\ \mathrm{has}\ \mathrm{def}\ \mathtt{this}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{S}})\mathtt{\{c\}} : \mathtt{U} = \ldots \\ \Gamma' = \Gamma, \overline{\mathtt{X}} : \mathtt{type}, \mathtt{this} : \mathtt{C}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \overline{\mathtt{V}} \mathtt{==} \overline{\mathtt{X}} \\ \Gamma'' = \Gamma, \overline{\mathtt{X}} : \mathtt{type}, \mathtt{this} : \mathtt{U}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \overline{\mathtt{V}} \mathtt{==} \overline{\mathtt{X}} \\ \sigma(\Gamma') \vdash_{\mathcal{C}} \mathtt{c} \\ \sigma(\Gamma') \vdash_{\mathcal{C}} \overline{\mathtt{T}} \mathtt{<:} \overline{\mathtt{S}} \\ \sigma(\Gamma'') \vdash_{\mathcal{C}} inv(\mathtt{C}),\end{array}}{\Gamma \vdash \mathtt{new\ C}[\overline{\mathtt{V}}](\overline{\mathtt{e}}) : \exists \overline{\mathtt{X}} : \mathtt{type}, \mathtt{this} : \mathtt{C}, \overline{\mathtt{x}} : \overline{\mathtt{T}}.\ \mathtt{U}} \quad (\textsc{T-new})$$

Figure 12. Typing rules

TODO: type parameters!

Now we consider the rule for method invocation. Assume that in a type environment Γ the expressions $\mathtt{e}_0,\ldots,\mathtt{e}_n$ have the types $\mathtt{T}_0,\ldots,\mathtt{T}_n$. Since the actual values of these expressions are not known, we shall assume that they take on some fixed but unknown values $\mathtt{z}_0,\ldots,\mathtt{z}_n$ of types $\mathtt{T}_0,\ldots,\mathtt{T}_n$. Now, for $\mathtt{z}_0$ as receiver, let us assume that the type $\mathtt{T}_0$ has a method named m with signature $[\overline{\mathtt{Z}}](\overline{\mathtt{z}}:\overline{\mathtt{S}})\{\mathtt{c}\} \rightarrow \mathtt{U}$ (Let $\mathtt{T}_0 = \mathtt{C\{d\}}$. If there is no method named m for the class C then this method invocation cannot be type-checked. Without loss of generality, we may assume that the type parameters of this method are named $\mathtt{Z}_1,\ldots,\mathtt{Z}_k$, and the value parameters are named $\mathtt{z}_1,\ldots,\mathtt{z}_n$ since we are free to choose variable names as we wish.) Now, for the method to be invokable, it must be the case that the types $\mathtt{T}_1,\ldots,\mathtt{T}_n$ are subtypes of $\mathtt{S}_1,\ldots,\mathtt{S}_n$. (Note that there may be no occurrences of this in $\mathtt{S}_1,\ldots,\mathtt{S}_n$—they have been replaced by $\mathtt{z}_0$.) Further, it must be the case that for these parameter values, the constraint c is entailed. Given all these assumptions it must be the case that the return type is U, with all the parameters $\mathtt{z}_0,\ldots,\mathtt{z}_n$ existentially quantified.

6.7.3 Class OK judgment

The following rule is modified from what we had in the paper to ensure that all the types are well-formed (under the assumption this :C). Note that the variables $\overline{\mathtt{x}}$ are permitted to occur in the types $\mathtt{T}_0,\overline{\mathtt{T}}$, hence their typing assertions must be added to Γ.

$$\frac{\begin{array}{c}\Gamma = \mathtt{this}:\mathtt{C\{self==this},\mathit{inv}(\mathtt{C})\},\overline{\mathtt{x}}:\overline{\mathtt{T}}\{\mathtt{self==}\}\overline{\mathtt{x}}\},\mathtt{c}\\ \Gamma \vdash \mathtt{e}:\mathtt{U}\\ \sigma(\Gamma)\vdash_{\mathcal{C}} \mathtt{U}\texttt{<:}\mathtt{T}\end{array}}{\mathtt{def}\ \mathtt{m}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}:\mathtt{T} = \mathtt{e}\ \text{OK in}\ \mathtt{C}} \quad (\text{METHOD OK})$$

This rule did not exist in our submission. This is necessary to ensure that the types of fields are well-formed.

$$\frac{\mathtt{this}:\mathtt{C},\mathtt{c}\vdash \mathtt{T}:\mathtt{type}}{\mathtt{val}\ \mathtt{f\{c\}}:\mathtt{T}\ \text{OK in}\ \mathtt{C}} \quad (\text{FIELD OK})$$

This rule is now modified to ensure that all the types and methods in the body of the class are well-formed.

$$\frac{\begin{array}{c}K\ \text{OK in}\ \mathtt{C}\\ \overline{\mathtt{M}}\ \text{OK in}\ \mathtt{C}\\ \overline{\mathtt{F}}\ \text{OK in}\ \mathtt{C}\\ \mathtt{this}:\mathtt{C}\vdash \mathtt{T}:\mathtt{type}\end{array}}{\mathtt{C}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{ext}\ \mathtt{T}\ \{\ \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{F}}\ \}\ \text{OK}} \quad (\text{CLASS OK})$$

TODO: method overriding

6.7.4 Subtype judgment

$$\frac{\sigma(\Gamma)\vdash_{\mathcal{C}} \mathtt{T}_1\texttt{<:}\mathtt{T}_2}{\Gamma\vdash \mathtt{T}_1\texttt{<:}\mathtt{T}_2}$$

7. Constraint solver

The goal of the constraint solver is to check an assertion $\overline{\mathtt{c}}\vdash_{\mathcal{C}} \mathtt{d}$.

We add the following rules to allow type arguments to calls to be omitted.

$$\frac{\begin{array}{c}\overline{\mathtt{Y}}\ \text{fresh}\\ \Gamma,\overline{\mathtt{Y}}:\mathtt{type}\vdash \mathtt{e}_0.\mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{e}}):\mathtt{T}\end{array}}{\Gamma\vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}):\mathtt{T}} \quad (\text{T-INVK-INFERRED})$$

$$\frac{\begin{array}{c}\overline{\mathtt{Y}}\ \text{fresh}\\ \Gamma,\overline{\mathtt{Y}}:\mathtt{type}\vdash \mathtt{new}\ \mathtt{C}[\overline{\mathtt{Y}}](\overline{\mathtt{e}}):\mathtt{T}\end{array}}{\Gamma\vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}):\mathtt{T}} \quad (\text{T-NEW-INFERRED})$$

7.1 Constraint representation

Represent a constraint as a graph G. Each node represents a constraint term for a value or a type. The node for a path p is written v_p; the node for a type T is written V_T. There are four kinds of edges:

1. undirected equivalence edges, $v_p \sim v_q$ and $V_S \sim V_T$,
2. type edges, $v_p \mapsto_{\mathtt{type}} V_T$,
3. tree edges, $v_p \mapsto_f v_{p.f}$ and $v_p \mapsto_X V_{p.X}$, and
4. flow edges, $V_S \rightarrow V_T$.

First, each constraint term is mapped to a node in the graph as follows. Associate each term t with a node v_t. For each access path p.x, add a tree edge $v_{\mathtt{p}} \mapsto_{\mathtt{x}} v_{\mathtt{p.x}}$. For each path type p.X, add a tree edge $v_{\mathtt{p}} \mapsto_{\mathtt{X}} V_{\mathtt{p.X}}$. For each atomic formula $\mathtt{f}(\overline{\mathtt{t}})$, add the tree edge $v_{\mathtt{f}(\overline{\mathtt{t}})} \mapsto_i v_{t_i}$ for all i. If term t has type T, add $v_t \mapsto_{\mathtt{type}} V_{t.\mathtt{type}}$ and add $V_T \sim V_{t.\mathtt{type}}$ to G.

Type nodes are sets of classes.

Next, constraints are incorporated into the graph:

- For constraint p==q, add $v_{\mathtt{p}} \sim v_{\mathtt{q}}$ to G.
- For constraint S==T, add $V_{\mathtt{S}} \sim V_{\mathtt{T}}$ to G.
- For constraint S<:T, add $V_{\mathtt{S}} \rightarrow V_{\mathtt{T}}$ to G.

7.2 Solving

A flow-path is a path that follows flow and equivalence edges only. A type-path is a path that follows type and equivalence edges only.

Now, we saturate: If there is a type-path $v_t \mapsto^*_{\mathtt{type}} V_{\mathtt{C\{c\}}}$, add $c[t/\mathtt{self}]$ to the worklist.

Can saturate lazily when doing a lookup. EXCEPT: a type may have an arbitrary constraint `C{self.x==3 && y > 7}`, so affect is non-local EXCEPT: c is x.f==... with x: Cc need to avoid infinite loop

To check:

- To check constraint p==q, check if $v_{\mathtt{p}} \sim^* v_{\mathtt{q}}$.
- To check constraint S<:T, check if there is a flow-path from $V_{\mathtt{S}}$ to $V_{\mathtt{T}}$. This requires checking entailment of the type constraints and adding more edges to the graph. (XXX details!) Add the flow edge to memoize.

8. 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 [1, 2] and the implementation of PolyJ [6]. 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[1]

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

8.1 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 13. It contains most of the features of generics that have to be translated.

8.2 Eliminating method type parameters

8.3 Translation to Java

8.4 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.

9. Discussion

todo: Move some of this to Section 2

9.1 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.

9.2 Wildcards

Wildcards in Java [18, 39] were motivated by the following example (rewritten in X10 syntax) from [39]. 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:

```
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 13. Code to translate

```
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] = { .
}
```

9.3 Proper abstraction

Consider the following example adapted from [39]:

```
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.

10. Conclusions

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

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

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

[4] David Aspinall and Martin Hofmann. *Dependent Types*, chapter 2. MIT Press, 2004.

[5] 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.

[6] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, 1997.

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

[12] ECMA. ECMA-334: C# language specification, June 2006. http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf.

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

[14] 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.

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

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

[17] 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.

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

[19] 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.

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

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

[22] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented progr amming. pages 397–406, October 1989.

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

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

[25] Andrew Myers and Barbara Liskov. Efficient implementation of parameterized types despite subtyping. Technical Report Thor Note 9, MIT LCS, June 1994.

[26] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software extension. In *Proceedings of the 2006 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 21–36, Portland, OR, October 2006.

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

[28] 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.

[29] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. pages 146–159, Paris, France, January 1997.

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

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

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

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

[34] 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.

[35] V. Saraswat et al. Report on the programming language X10. Technical report, IBM T.J. Watson Research Center, 2006.

[36] 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.

[37] Kresten Krab Thorup. Genericity in Java with virtual types. Number 1241, pages 444–471, 1997.

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

[39] 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.

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

[41] 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.