Constrained Kinds

Olivier Tardieu
IBM Research
tardieu@us.ibm.com

Nathaniel Nystrom
University of Lugano
nate.nystrom@usi.ch

Igor Peshansky
Google
igorp@acm.org

Vijay Saraswat
IBM Research
vsaraswa@us.ibm.com

Abstract

Modern object-oriented languages such as X10 require a rich framework for types capable of expressing both value-dependency and genericity, and supporting pluggable, domain-specific extensions.

In earlier work, we presented a framework for *constrained types* in object-oriented languages, parametrized by an underlying constraint system. Types are viewed as formulas `C{c}` where `C` is the name of a class or an interface and `c` is a constraint on the immutable instance state (the *properties*) of `C`. Constraint systems are a very expressive framework for partial information. Many (value-)dependent type systems for object-oriented languages can be viewed as constrained types.

This paper extends the constrained types approach to handle *type-dependency* ("genericity"). The key idea is to introduce *constrained kinds*: in the same way that constraints on values can be used to define constrained types, constraints on types can define constrained kinds.

We develop a core programming language with constrained kinds. Generic types are supported by introducing type variables—literally, variables with "type" `Type`—and permitting programs to impose subtyping and equality constraints on such variables. We formalize the type-checking rules and establish soundness.

While the language now intertwines constraints on types and values, its type system remains parametric in the choice of the value constraint system (language and solver). We demonstrate that constrained kinds are expressive and practical and sketch possible extensions with a discussion of the design and implementation of X10.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications—object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, constraints; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—object-oriented constructs, type structure

General Terms Design, Languages, Theory

Keywords types; generics; constraints; X10

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

1. Introduction

Dependent types [9, 31, 49] offer opportunities for detecting programming errors statically and for eliminating costly array bounds, null dereference, or other run-time checks. The X10 programming language takes advantages of *constrained types* [37]—a form of dependent types—to provide an open-ended, user-extensible framework in which to specify and enforce desirable properties of data structures statically.

Generic types, types such as `List<T>` in Java that are parametrized by other types, are widely established [5, 12, 18, 25, 35, 36, 44], and are vital for implementing type-safe, reusable libraries, especially collections classes.

X10, like Java, initially had no support for genericity. The subtle issues encountered when designing and implementing a generic type system for X10 exposed the need for a formal framework in which to explore the design space and to reason about fundamental issues of soundness and expressivity. As a result, this paper develops the framework of *constrained kinds*, unifying constrained types and generic types.

1.1 Constrained Types

In X10, a normal class type `C` is enriched to permit a *constrained type* `C{c}` where `c` is a constraint on the immutable fields, or *properties*, of the class `C` as well as any final variables and constants in scope. Constraints are drawn from a constraint language that, syntactically, is a subset of the boolean expressions of X10. For instance, `Point{self.rank==n}` is a type satisfied by any `n`-dimensional point, that is, any instance of `Point` whose `rank` property is `n`. Here, `n` is a final variable whose value may be unknown statically. In a constraint, `self` refers to a value of the base type being constrained, in this case `Point`.

Constraints may be used to specify class invariants and conditions on the accessibility of methods. For instance, the

euclidian `distance` method of the `Point` class requires that the ranks of the points be the same:

```
class Point(rank:Int) {
  def distance(p:Point){this.rank==p.rank} ...
}
```

Therefore the X10 compiler is able to flag and reject programs trying to compute the distance between a 2-d point and a 3-d point. Reciprocally, if two points are known statically to have the same rank, even if the actual rank itself is not known statically, the compiler is able to type check the distance method invocation.

The key idea behind X10's approach is that the type-checking rules can be decoupled from the machinery of constraints. By varying the constraint language and solver, one can tune the X10 type system to the specific needs of a particular application domain, with confidence that the result is sound.

1.2 Generic Types

Generic types are essential for implementing type-safe, extensible collections libraries. In Java, for example, generic types let programmers distinguish different types of lists such as lists of integers `List<Integer>` or lists of lists of strings `List<List<String>>`. In general, Java types can be parametrized with other types. The compiler keeps track of the type parameters and guards against mismatches.

In other languages, including X10, genericity benefits extend beyond static safety. For instance, X10 permits the declaration of struct types reminiscent of C structs. The runtime representation of an X10 array is customized to the content of the array: structs are inlined into arrays whereas arrays of objects are implemented as arrays of pointers.

1.3 Constrained Kinds

This paper lays out a framework to extend constrained types to handle type genericity. The general outline of the approach is to introduce type variables and a vocabulary of constraints over types. These constraints on types are used to specify *constrained kinds* in the same way that constraints over values are used to specify *constrained types*.

In different programming languages, type variables are introduced as *type parameters* (cf. Java [18]) or as *type members* (cf. BETA [27]). In the framework described in this paper, a *type variable* can be declared as a property of a class or as a method parameter with type `Type`. For example, one can declare an `Array` class with a type property `X` as well as an integer property `rank`:

```
class Array(X:Type,rank:Int) { ... }
```

Within the scope of its declaration, a type variable can be used wherever a type can (e.g., to specify the type of a value property or method parameter, or as the target of a cast).

Like value properties, type properties can be used in constrained types through the variable `self`. With the above `Array` declaration, the type of an array of integers, say, can be written as `Array{self.X==Int}`.

```
1  public class Array[T](rank:Int) {
2    private val raw:Block[T]; // raw memory
3    private val size:Int;
4    ...
5
6    /* rank 1 constructors */
7    public def this(size:Int,init:T) {
8      ...
9      raw = Block.allocateUninitialized[T](size);
10     for (i in 0..(size-1)) raw(i) = init;
11   }
12
13   public def this(size:Int){T haszero} {
14     ...
15     raw = Block.allocateZeroed[T](size);
16   }
17
18   /* getter */
19   public def get(p:Point){rank==p.rank}:T {
20     return raw(offset(p));
21   }
22
23   /* rank 1 getter */
24   public def get(i:Int){rank==1}:T {
25     return raw(i);
26   }
27
28   public def sum(){T<:Arithmetic[T]}:T {
29     var acc:T = raw(0);
30     for (i in 1..(size-1)) acc += raw(i);
31     return acc;
32   }
33 }
```

Figure 1. X10 array class.

To make constrained kinds expressive, a suitable vocabulary of constraints over types must be chosen. In the context of nominal object-oriented languages such as Java and X10, types are equipped with a partial order (the subtyping relation) generated from the user program through the "`extends`" and "`implements`" relationships. This structure motivates a constraint system in which, for a type variable `X` one can assert the constraint `X<:T`. A valuation (a mapping from variables to types) realizes this constraint if it maps `X` to a type that is a subtype of `T`. Constrained kinds can therefore express bounds on type variables similar to those in Java: `X:Type{self<:Number}` declares a type variable `X` which can only be bound to those types `S` that satisfy `S<:Number`.

1.4 Example

Figure 1 shows a fragment of an `Array` class in the X10 syntax. This class introduces a type parameter `T` in square brackets (see Section 5).

The fragment shows two constructors. The first constructor (lines 7–11) takes an array size and an initial value for the array elements; the second (lines 13–16) takes only a size. The second constructor constrains `T` with a `haszero` constraint, which holds when a type contains a zero value, that is, a value whose representation is a pattern of zero bits. This constraint allows fast allocation of arrays containing the zero value, but safely ensures a zero-backed array cannot be created when `T` is bound to a type such as `Object{self!=null}` with no zero value . Thus, to allocate an `Array[Object{self!=null}]`, the programmer must use the first constructor, passing in an explicit initial value.

The example also shows two getter methods: the first (lines 19–21) requires a point of matching rank, while the second (lines 24–26) allows a single integer to index into the array, but only if the array is of rank 1.

Finally, a `sum` method (lines 28–32) is defined, which uses a subtyping constraint to require that the method only be invoked on arrays of arithmetic types; that is, types providing the usual arithmetic operators.

The subtyping, `haszero`, and rank constraints all provide partial information about the types (both requirements and guarantees). The framework in this paper presents a unified formalism for constraints on both types and values such as the above. It permits programmers to provide more static information to the compiler to enable safer, more efficient libraries and to allow the compiler to generate faster code with fewer run-time checks.

1.5 Contributions

This paper develops the framework of constrained kinds, unifying constrained types and generic types. We present a formalization of these ideas through an extension of the development in our prior work on constrained types [37]. We summarize our contributions:

- We present a core "featherweight" calculus for constrained kinds, FXG, parametrized on an underlying value constraint system. The calculus can express `X<:T` constraints on type variables and offers the programmer a unified view of value and type dependency.
- We formalize its type system and prove type soundness. The type-checking rules cleanly disentangle type constraints from value constraints, directly solving type constraints while funneling value constraints to the constraint system.
- We show how the framework can be extended in a simple, methodical fashion to handle additional constraints such as arithmetic constraints or structural subtyping constraints.
- While our focus here is on a formal framework for constrained kinds, we address issues of practicality in the context of X10. A version of the framework forms the core of the X10 type system. To realize the framework, several design choices were made that restrict the expressiveness in favor of efficiency, ease of use, and implementation. We discuss how design alternatives such as the choice of type parameters versus type members, use-site versus definition-site variance, and nominal versus structural subtyping constraints can be expressed in the framework.

The high-level design goal of FXG is to keep its type-checking rules as simple as possible while delegating the bulk of the hard work to the constraint system. From an operational perspective, the FXG type checker asks a constraint solver whether a given context entails a given constraint and accepts or rejects programs based on the answers.

As shown in [37], this modular approach helps at many levels including soundness, expressivity, and performance. First, soundness is easier to ensure as it results in large part from the soundness of the constraint system itself. Second, the requirements on the constraint system are minimal, making it easy to explore the design space and vary the static guarantees, annotation overhead, compile-time and run-time costs, etc. Finally, the type checker performance will benefit from highly optimized constraint solvers.

Outline. The rest of the paper is organized as follows. Section 2 introduces FXG with its formal semantics and type-checking rules. We prove type soundness in Section 3 and discuss formal extensions of FXG in Section 4. Section 5 follows with a discussion of design choices and of how the framework is realized concretely in X10. While this section builds on the formalism introduced in the previous sections, much of this discussion should be understandable after a quick read of Section 2. Related work is discussed in Section 6. Section 7 concludes the paper.

2. The FXG Language

In this section, we introduce a core formal programming language, FXG, unifying constrained types and generic types. We describe its syntax, operational semantics, and type system.

Our language models many but not all of the relevant features of X10. Following FJ [20], it does not account for mutable state. Objects once constructed are immutable. All variables are final. We model classes but not interfaces, method overriding but not method overloading, default constructors but not user-defined constructors. None of these restrictions impacts the formalism, its soundness, or expressivity in an essential way. We will revisit these decisions when we discuss X10 in Section 5.

2.1 Syntax

The grammar for FXG is shown in Figure 2. The syntax is essentially that of X10. We use $\overline{x}$ to denote a possibly empty list $x_1, \ldots, x_n$ and $\bullet$ to denote the empty list. A program P is a finite set of class declarations $\overline{\mathsf{L}}$.

(Class declaration)	L	::=	$\texttt{class C}(\overline{\texttt{f}}{:}\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends C}\ \{\ \overline{\texttt{M}}\ \}$
(Method declaration)	M	::=	$\texttt{def m}(\overline{\texttt{x}}{:}\overline{\texttt{T}})\{\texttt{c}\}{:}\texttt{T} = \texttt{e};$
(Path)	p	::=	$\texttt{x} \mid \texttt{p.f}$
(Kind)	K	::=	$\texttt{Type}\{\texttt{c}\}$
(Type)	R, S, T	::=	$\texttt{K} \mid \texttt{T}_\emptyset$ where $\texttt{T}_\emptyset ::= \texttt{C}\{\texttt{c}\} \mid \texttt{p}$
(Expression)	e	::=	$\texttt{x} \mid \texttt{new C}(\overline{\texttt{e}}) \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{e as T}_\emptyset \mid \texttt{C}\{\texttt{c}\}$
(Constraint term)	t	::=	$\texttt{x} \mid \texttt{new C}(\overline{\texttt{t}}) \mid \texttt{t.f} \mid \texttt{C}\{\texttt{c}\}$
(Value constraint)	$\texttt{c}_\emptyset$	::=	$\texttt{true} \mid \texttt{false} \mid \texttt{t} == \texttt{t} \mid \texttt{c}_\emptyset, \texttt{c}_\emptyset$
(Constraint)	c	::=	$\texttt{T}_\emptyset <: \texttt{T}_\emptyset \mid \texttt{c}_\emptyset \mid \texttt{c}, \texttt{c}$
(Value)	v, w	::=	$\texttt{new C}(\overline{\texttt{v}}) \mid \texttt{C}\{\texttt{c}\}$ where c contains no variable other than possibly self

C, D range over class names, f, g over field names, m over method names, x, y, z over variable names.

Figure 2. FXG productions.

$$\frac{\mathsf{fields}(\texttt{C}) = \overline{\texttt{f}} : \overline{\texttt{T}}}{\texttt{new C}(\overline{\texttt{v}}).\texttt{f}_i \to \texttt{v}_i} \quad \text{(R-FIELD)} \qquad \frac{\texttt{e}_i \to \texttt{e}_i'}{\texttt{new C}(\texttt{v}_1, \ldots, \texttt{v}_{i-1}, \texttt{e}_i, \ldots, \texttt{e}_n) \to \texttt{new C}(\texttt{v}_1, \ldots, \texttt{v}_{i-1}, \texttt{e}_i', \ldots, \texttt{e}_n)} \quad \text{(RC-NEW-ARG)}$$

$$\frac{\texttt{e} \to \texttt{e}'}{\texttt{e.f} \to \texttt{e}'.\texttt{f}} \quad \text{(RC-FIELD)} \qquad \frac{\mathsf{method}(\texttt{C}, \texttt{m}) = \texttt{m}(\overline{\texttt{x}}{:}\overline{\texttt{T}})\{\texttt{c}\}{:}\texttt{R} = \texttt{e}}{\texttt{new C}(\overline{\texttt{v}}).\texttt{m}(\overline{\texttt{w}}) \to \texttt{e}[\texttt{new C}(\overline{\texttt{v}}), \overline{\texttt{w}}/\texttt{this}, \overline{\texttt{x}}]} \quad \text{(R-INVK)}$$

$$\frac{\texttt{e} \to \texttt{e}'}{\texttt{e.m}(\overline{\texttt{e}}) \to \texttt{e}'.\texttt{m}(\overline{\texttt{e}})} \quad \text{(RC-INVK-RECV)} \qquad \frac{\texttt{e}_i \to \texttt{e}_i'}{\texttt{v.m}(\texttt{w}_1, \ldots, \texttt{w}_{i-1}, \texttt{e}_i, \ldots, \texttt{e}_n) \to \texttt{v.m}(\texttt{w}_1, \ldots, \texttt{w}_{i-1}, \texttt{e}_i', \ldots, \texttt{e}_n)} \quad \text{(RC-INVK-ARG)}$$

$$\frac{\texttt{e} \to \texttt{e}'}{\texttt{e as T} \to \texttt{e}' \texttt{ as T}} \quad \text{(RC-CAST)} \qquad \frac{\vdash \texttt{new C}(\overline{\texttt{v}}){:}\texttt{S}, \texttt{S} <: \texttt{T}}{\texttt{new C}(\overline{\texttt{v}}) \texttt{ as T} \to \texttt{new C}(\overline{\texttt{v}})} \quad \text{(R-CAST)}$$

Figure 3. Operational semantics.

$$\mathsf{fields}(\texttt{Object}) = \bullet \quad \text{(L-FIELD-B)}$$

$$\frac{\texttt{class C}(\overline{\texttt{f}}{:}\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends C}'\ \{\ \overline{\texttt{M}}\ \} \qquad \mathsf{fields}(\texttt{C}') = \overline{\texttt{f}'}{:}\overline{\texttt{T}'}}{\mathsf{fields}(\texttt{C}) = \overline{\texttt{f}'}, \overline{\texttt{f}}{:}\overline{\texttt{T}'}, \overline{\texttt{T}}} \quad \text{(L-FIELD-I)}$$

$$\frac{\texttt{class C}(\overline{\texttt{f}}{:}\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends C}'\ \{\ \overline{\texttt{M}}\ \} \qquad \texttt{def m}(\overline{\texttt{x}}{:}\overline{\texttt{T}'})\{\texttt{c}'\}{:}\texttt{R} = \texttt{e} \in \overline{\texttt{M}}}{\mathsf{method}(\texttt{C}, \texttt{m}) = \texttt{m}(\overline{\texttt{x}}{:}\overline{\texttt{T}'})\{\texttt{c}'\}{:}\texttt{R} = \texttt{e}} \quad \text{(L-METHOD-B)}$$

$$\frac{\texttt{class C}(\overline{\texttt{f}}{:}\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends C}'\ \{\ \overline{\texttt{M}}\ \} \qquad \mathsf{method}(\texttt{C}', \texttt{m}) = \texttt{m}(\overline{\texttt{x}}{:}\overline{\texttt{T}'})\{\texttt{c}'\}{:}\texttt{R} = \texttt{e} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathsf{method}(\texttt{C}, \texttt{m}) = \texttt{m}(\overline{\texttt{x}}{:}\overline{\texttt{T}'})\{\texttt{c}'\}{:}\texttt{R} = \texttt{e}} \quad \text{(L-METHOD-I)}$$

Figure 4. Fields and methods.

1. Classes. A class has a name C, final fields $\overline{\texttt{f}}$ with types $\overline{\texttt{T}}$, a superclass C, methods $\overline{\texttt{M}}$, and a class invariant c—a constraint on the fields valid for all instances of the class. Like any other constraint, the class invariant may be omitted. An omitted constraint simply stands for the `true` constraint.

Class names C range over the declared classes in P and `Object`. As usual, we assume the classes of a program have distinct names, which are also distinct from `Object` and `Type`. The class `Object` is implicitly declared, has no fields or methods, and does not extend any other class.

We define the inheritance relation—C *inherits from* C′—as the transitive closure of the `extends` relation. We assume that the inheritance graph has no cycles or self loops; hence, it is a tree with class `Object` as its root.

2. Fields and constructors. The fields of a class C are the union of the fields of the superclasses and the fields declared in C.

```
class C(x:Object) extends Object {}
class D(y:Type) extends C {}
class E(z:y) extends D {}
```

In this example, C has one field named `x`, D has two named `x` and `y` in this order, and E has three.

The fields are ordered by their declaration with the fields of the superclass coming before the fields of the declared class. We assume the names of the fields of a class to be distinct from one another.

A field may be a type variable (e.g., the `y` field of D). The type of a field may involve fields already declared. Here, field `z` is declared with type `y`.

Each class has an implicit default constructor that takes one argument for each field of the class, in order, and initializes the fields with these arguments. The `Object` class has a 0-ary constructor.

3. Methods. Methods are introduced with the `def` keyword. A method has formals $\overline{\texttt{x}}$ with types $\overline{\texttt{T}}$ and return type T. The method guard c is to be thought of as an additional condition that must be satisfied by the receiver and the arguments of the method call. The body of a method is an expression e.

We assume that the formals of a method have distinct names, none of which are `this`. We do not consider method overloading: we assume each class declares at most one method with a given name.

The type of a formal may involve a formal declared to its left as well as the fields of the enclosing class. For instance, the `distance` method of the `Point` class of Section 1 could also be declared:

```
def distance(p:Point{this.rank==self.rank}) ...
```

Here, `this.rank` denotes the `rank` of method receiver and `self.rank` denotes the `rank` of `p`. Ultimately this constraint on the type of `p` or the method guard as defined in Section 1 impose the same restrictions on the method's applicability.

4. Variables and paths. The variables in scope in the body of a method are the method formals $\overline{\texttt{x}}$ and the implicit receiver `this`. Paths, e.g, `x.f.g`, are chains of field selections starting with a variable.

5. Types, kinds, and type variables. Types in FXG are firstly *nominal types*: each class name C defines a type C. Informally, a value is of type C if it is an instance of the class C.[1]

Types include *constrained class types* $\texttt{C}\{\texttt{c}\}$ and *constrained kinds* $\texttt{Type}\{\texttt{c}\}$. If value v is of type $\texttt{T}\{\texttt{c}\}$ then it satisfies the constraint $\texttt{c}[\texttt{v}/\texttt{self}]$. Formals and fields declared with type $\texttt{Type}\{\texttt{c}\}$ are said to be *type variables*.

Finally, there are *path types* p. We assume that paths used in type positions are type variables, hence the name path types.

We write $\texttt{T}_\emptyset$ for a type that is not a kind, that is, a constrained class type or a path type.

A path type is never a kind. If a formal or field is declared with path type p then it cannot be a type variable. In other words, its value cannot be a type. As a result, while type variables are not segregated from standard variables in FXG using the bracket notation of Java or X10, it is always possible to partition fields and formals as either type variables or standard variables by just looking at their declared types.

While a method's return type may be a kind, an invocation of such a method cannot appear in type position (i.e., as the type of a formal or a field, or as the target of a cast).

6. Values, expressions, and constraint terms. There are two sorts of values: object instances and constrained class types $\texttt{C}\{\texttt{c}\}$ where the only variable permitted in c is `self`. Formally, $\texttt{C}\{\texttt{c}\}$ binds variable `self` in c; a value $\texttt{C}\{\texttt{c}\}$ may not have free variables. Following FJ, we denote object instances by means of nested constructor calls, e.g., "`new Box(C,new C())`."

Expressions are built from variables in scope, field accesses, constructor calls, method invocations, casts (written $\texttt{e as T}_\emptyset$), and constrained class types. Casts for values of type `Type` such as "`C as Type{self==C}`" are unsupported due to a lack of compelling use cases. In FXG as in X10 or Java, C in constructor invocation $\texttt{new C}(\overline{\texttt{e}})$ must be a class name, not a type or type variable.

The set of constraint terms is a subset of the set of expressions and a superset of the set of values. It includes variables and constrained class types and is closed under field selection and object construction.

7. Constraints. Constraints are built from the conjunction of the constraints `true` and `false`, equality constraints $\texttt{t} == \texttt{t}$, and subtyping constraints $\texttt{T}_\emptyset <: \texttt{T}_\emptyset$.

We write $\texttt{c}_\emptyset$ for value constraints, that is, in this core language, equality constraints. Value constraints include equality constraints on types such as $\texttt{x} == \texttt{C}\{\texttt{c}\}$ (see Section 2.3).

[1] Formally, the type of an instance of class C is $\exists\overline{\texttt{x}}{:}\overline{\texttt{T}}.\ \texttt{C}\{\texttt{c}\}$ for some types $\overline{\texttt{T}}$ and constraint c, therefore a subtype of C.

A class invariant may only refer to the variable `this`. Moreover, it may only refer to `this` in field selection expressions. A method guard may refer to the receiver `this` and the formals $\overline{\mathtt{x}}$ of the method. In general, a constraint `c` in a constrained type `T{c}` may refer to the variable `self` in addition to the variables in scope where `self` refers to any value of the type `T` being constrained. In particular, a return type may refer to the receiver and the formals of the method as well as to the return value itself (i.e., `self`).

Constraints may be nested, for example:

```
Type{self<:Nat{self==new Zero()}}
```

Here the outer `self` corresponds to the type being constrained, the inner `self` to a value of that type.

2.2 Operational Semantics

The operational semantics, shown in Figure 3, is described as a reduction relation on expressions $\mathtt{e} \to \mathtt{e}'$. It enforces a strict left-to-right call-by-value evaluation order.

It uses two helper predicates defined in Figure 4. The fields predicate computes the list of fields of a given class. The method predicate returns the declaration of method `m` available in class `C`, if it exists. It is recursively defined as either the method `m` declared in class `C`, if any, or else as the method `m` available in the superclass of `C`, if any.

In rule R-INVK, we use $[\overline{\mathtt{x}}/\overline{\mathtt{y}}]$ to denote the substitution of the `y`'s by the `x`'s. This implicitly requires the two lists to have the same length, hence R-INVK ensures that the method call has the correct number of arguments.

Unsurprisingly, the dynamic semantics only depends on the type system via the rule R-CAST. In short, the rule specifies that `new` $\mathtt{C}(\overline{\mathtt{v}})$ can be cast to type `T` iff `new` $\mathtt{C}(\overline{\mathtt{v}})$ can be shown to have type `S` where `S` is a subtype of `T`. It is worth noting that, in theory, casts require run-time invocations of the static type system, which may involve constraint solving (see Section 5.4).

Except for casts, constraints are irrelevant to the dynamic semantics. We will establish that there is no need for run-time checking of method guards or class invariants for a well-typed program. In essence, every variable with a constrained type `T{c}` is guaranteed to be bound to a value that satisfies `c` at run time.

2.3 Constraint System

The FXG definition is parametrized by a *value constraint system* $\mathcal{X}$. This constraint system is required to have the predicates and terms of our constraint language with an adequate interpretation. It may have other predicates and terms whose interpretation is left unconstrained.

Formally, $\mathcal{X}$ is required to have terms `t` of the form "$\mathtt{C}(\overline{\mathtt{f}} = \overline{\mathtt{t}})$", "`t.f`", "`C{c}`", and an equality predicate "==" on such terms. We map FXG constraint terms "`new` $\mathtt{C}(\overline{\mathtt{t}})$" to $\mathcal{X}$ terms "$\mathtt{C}(\overline{\mathtt{f}} = \overline{\mathtt{t}})$" so as to capture field names in the term itself (see the definition of constraint projections below).

The entailment relation for $\mathcal{X}$ must respect the interpretation of (a) $\mathtt{C}(\overline{\mathtt{f}} = \overline{\mathtt{t}})$ as a finite tree with root labeled with `C`, ith branch labeled with $\mathtt{f}_i$ and leading to $\mathtt{t}_i$, and (b) `t.f` as selection of the child labeled `f` for the tree `t`.[2]

Equality is reflexive, symmetric, transitive, and a congruence w.r.t. field selection: if `x == y` then `x.f == y.f`. Moreover $\mathtt{C}(\overline{\mathtt{f}} = \overline{\mathtt{t}}) == \mathtt{C}'(\overline{\mathtt{f}}' = \overline{\mathtt{t}}')$ iff the class names and field names are identical and $\overline{\mathtt{t}} == \overline{\mathtt{t}}'$. Using object-oriented terminology, equality is structural.

Terms of the form `C{c}` are just that, terms, with no semantics or structure as far as $\mathcal{X}$ is concerned. Intuitively, we want $\mathcal{X}$ to solve term equivalence constraints irrespective of the sort of the terms, but subtyping constraints will be handled outside of $\mathcal{X}$.

1. Inconsistent constraints. A type `T` of may be *inconsistent* due to *inconsistent constraints*, that is, there exists no value of type `T`. This may be due to value constraints as in the type "`C{self==new C(),self==new D()}`" or type constraints as in the kind "`Type{self<:C,self<:D}`."

While it makes sense to report inconsistent types, class invariants, or guards to the programmer (see Section 5.6), inconsistent constraints are not a first-order concern of FXG. Indeed, as long as methods with inconsistent guards cannot be invoked, objects with inconsistent invariants cannot be constructed, or casts to inconsistent types cannot succeed, type soundness can be established.

Inconsistent subtyping constraints however complicate things because they essentially bring back multiple inheritance to FXG despite the initial single-inheritance assumption. For instance, if `x` has type `T` and `T` has type "`Type{self<:C,self<:D}`" then both the methods of `C` and `D` are available on `x`, which lead to ambiguities. Therefore, we adopt a mixed approach in FXG where we disallow inconsistent subtyping constraints, but consider inconsistent value constraints harmless. We formalize this shortly.

2.4 Type System

Type checking FXG programs involves judgments about constraint entailment, subtyping, member lookup, and typing per se. Below, `i` stands for the name of a field of method and `I`, `I'` for field or method signatures:

$$\mathtt{I}, \mathtt{I}' ::= \mathtt{C.f:T} \mid \mathtt{C.m}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}:\mathtt{R}$$

Formally, we consider:

1. Constraint entailment:
   $\Gamma \vdash \mathtt{c}_0$ environment Γ entails value constraint $\mathtt{c}_0$
2. Subtyping:
   $\Gamma \vdash \mathtt{S <: T}$ the type `S` is a subtype of type `T` in Γ
   $\Gamma \vdash \mathtt{x :: T}$ the type of `x` is a subtype of type `T` in Γ
3. Member lookup:
   $\Gamma \vdash \mathtt{T.i} \longrightarrow \mathtt{I}$ `T.i` resolves to field or method `I` in Γ

[2] A complete axiomatization of the algebra of finite trees is provided in [29].

$\Gamma \vdash \mathtt{T.i} \Longrightarrow \mathtt{I}$ $\quad$ $\mathtt{T.i}$ ambiguously resolves to $\mathtt{I}$ in Γ

$\mathtt{I} \ll \mathtt{I}'$ $\quad$ $\mathtt{I}$ overrides $\mathtt{I}'$

4. Typing:

$\Gamma \vdash \mathtt{e} : \mathtt{T}$ $\quad$ expression $\mathtt{e}$ has type $\mathtt{T}$ in Γ

$\vdash \mathtt{def\ m}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\}:\mathtt{R} = \mathtt{e}$ OK in $\mathtt{C}$ $\quad$ method $\mathtt{m}$ in class $\mathtt{C}$ is well typed

$\vdash \mathtt{class\ C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends\ C}'\ \{\ \overline{\mathtt{M}}\ \}$ OK $\quad$ class $\mathtt{C}$ is well typed

A program $\mathtt{P}$ is well typed iff all its classes are well typed.

In the definition of these judgments, the program source is an *axiom* of the deduction system. Therefore, judgments implicitly depend on the program $\mathtt{P}$ under consideration.

The environment Γ is a finite, possibly empty sequence of variable declarations $\mathtt{x} : \mathtt{T}$ and constraints $\mathtt{c}$. In the type system specification, we often need to partially specify the environment. In particular, we write "$\Gamma, \mathtt{x} : \mathtt{T}, \Delta$" for an environment that declares $\mathtt{x}$ with type $\mathtt{T}$ and is otherwise unconstrained.

Before we specify each of these judgments in turn, we discuss well-formedness, existential types, and class invariants.

1. Well-formedness. A constraint, type, or expression is well formed in environment Γ iff its free variables are declared in Γ. Given the scoping rules we sketched earlier in this section, the rules of well-formedness are straightforward. We omit them.

An environment Γ is well formed iff each constraint and type in Γ is well formed w.r.t. to the sequence of variable declarations to its left. This includes the current variable declaration. For example, "$\mathtt{x} : \mathtt{Object}\{\mathtt{self} == \mathtt{x}\}$" is a well-formed environment.

It is worth keeping in mind that for $\Gamma, \mathtt{x} : \mathtt{y}, \Delta$ to be well-formed it must be that $\mathtt{y}$ is declared as a type variable in Γ with type $\mathtt{Type}\{\mathtt{c}\}$ for some $\mathtt{c}$. Moreover, $\mathtt{x}$ is not a type variable.

We assume well-formedness through the rest of the paper.

2. Existential types. The family of types $\mathtt{E}$ we consider in the type system is richer than source types $\mathtt{T}$ as defined in Figure 2.

$$\mathtt{E} ::= \mathtt{T} \mid \mathtt{E}\{\mathtt{c}\} \mid \exists \mathtt{x}:\mathtt{E}.\ \mathtt{E}$$

First, we permit constraints on path types, for example:

$$\mathtt{x} : \mathtt{Type}, \mathtt{y} : \mathtt{x} \vdash \mathtt{y} : \mathtt{x}\{\mathtt{self} == \mathtt{y}\}$$

Second, types are closed under existential quantification. Informally, a value $\mathtt{v}$ is of type $\exists \mathtt{x}:\mathtt{S}.\ \mathtt{T}$ if there exists some value $\mathtt{w}$ of type $\mathtt{S}$ such that $\mathtt{v}$ is of type $\mathtt{T}[\mathtt{w}/\mathtt{x}]$.

We use existential types to conveniently accumulate constraints in chains of existentials. For instance, if class $\mathtt{C}$ has a field $\mathtt{f}$ with type $\mathtt{Object}$, we can establish

$$\mathtt{x} : \mathtt{C} \vdash \mathtt{x.f} : \exists \mathtt{y}:\mathtt{C}\{\mathtt{self} == \mathtt{x}\}.\ \mathtt{Object}\{\mathtt{self} == \mathtt{y.f}\}$$

While, as part of the type-checking rules, we could attempt to simplify this to "$\mathtt{x} : \mathtt{C} \vdash \mathtt{x.f} : \mathtt{Object}\{\mathtt{self} == \mathtt{x.f}\}$," we choose not to do so in the type system and rely on the constraint solver to coalesce the equality constraints. In our implementation however, we adopt an eager approach—no existentials—for performance reasons.

For simplicity, we keep using symbols $\mathtt{R}$, $\mathtt{S}$, and $\mathtt{T}$ to refer to types of the extended family.

3. Class invariants. A class invariant is intended to hold for any instance of that class. Therefore, in all judgments, if Γ declares a variable $\mathtt{x}$ of type $\mathtt{C}$ then the invariant for class $\mathtt{C}$ with $\mathtt{x}$ substituted for $\mathtt{this}$ is implicitly added to Γ.[3]

4. Constraint entailment. In general, a typing environment is expressed in terms that are outside of the domain of the constraint system, such as existential types, subtyping

The constraint projections σ and π are defined similarly. To save space, we introduce ω to abstract over them.

$$\omega \in \{\sigma, \pi\}$$

$$\begin{aligned}
\omega(\mathtt{x}) &= \mathtt{x} \\
\omega(\mathtt{new\ C}(\overline{\mathtt{t}})) &= \mathtt{C}(\overline{\mathtt{f}} = \omega(\overline{\mathtt{t}})) \text{ if } \mathsf{fields}(\mathtt{C}) = \overline{\mathtt{f}} : \overline{\mathtt{T}} \\
\omega(\mathtt{t.f}) &= \omega(\mathtt{t}).\mathtt{f} \\
\omega(\mathtt{C}\{\mathtt{c}\}) &= \mathtt{C}\{\mathtt{c}\}
\end{aligned}$$

$$\begin{aligned}
\omega(\mathtt{true}) &= \mathtt{true} \\
\omega(\mathtt{false}) &= \mathtt{false} \\
\omega(\mathtt{c}, \mathtt{c}') &= \omega(\mathtt{c}), \omega(\mathtt{c}')
\end{aligned}$$

$$\begin{aligned}
\omega(\mathtt{c}, \Gamma) &= \omega(\mathtt{c}), \omega(\Gamma) \\
\omega(\mathtt{x}:\mathtt{Type}, \Gamma) &= \omega(\Gamma) \\
\omega(\mathtt{x}:\mathtt{C}, \Gamma) &= \omega(\Gamma) \\
\omega(\mathtt{x}:\mathtt{y}, \Gamma) &= \omega(\Gamma) \\
\omega(\mathtt{x}:\mathtt{T}\{\mathtt{c}\}, \Gamma) &= \omega(\mathtt{x}:\mathtt{T}, \mathtt{c}[\mathtt{x}/\mathtt{self}], \Gamma) \\
\omega(\mathtt{x}:\exists \mathtt{y}:\mathtt{T}.\ \mathtt{S}, \Gamma) &= \omega(\mathtt{z}:\mathtt{T}, \mathtt{x}:\mathtt{S}[\mathtt{z}/\mathtt{y}], \Gamma)
\end{aligned}$$

$$\sigma(\mathtt{t} == \mathtt{t}') = \sigma(\mathtt{t}) == \sigma(\mathtt{t}') \qquad \pi(\mathtt{t} == \mathtt{t}') = \mathtt{true}$$

$$\sigma(\mathtt{S}_\emptyset <: \mathtt{T}_\emptyset) = \mathtt{true} \qquad \pi(\mathtt{S}_\emptyset <: \mathtt{T}_\emptyset) = \mathtt{S}_\emptyset <: \mathtt{T}_\emptyset$$

Figure 5. Constraint projections.

$$\frac{\sigma(\Gamma) \vdash \mathtt{c}_\emptyset \text{ in } \mathcal{X}}{\Gamma \vdash \mathtt{c}_\emptyset} \qquad \text{(X-PROJ)}$$

Figure 6. Constraint entailment.

[3] In contrast, we do not recursively add invariants for the fields of $\mathtt{x}$. It is the responsibility of the FXG programmer to include selected fragments of these invariants into the class invariant itself. See [37] for a discussion of the alternative.

$$\frac{\mathtt{x}\text{ fresh} \qquad \Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{x} :: \mathtt{T}}{\Gamma \vdash \mathtt{S} <: \mathtt{T}} \quad \text{(S-SUB)}$$

$$\frac{\mathtt{S} <: \mathtt{T} \in \pi(\Gamma, \mathtt{x}:\mathtt{S}, \Delta)}{\Gamma, \mathtt{x}:\mathtt{S}, \Delta \vdash \mathtt{x} :: \mathtt{T}} \quad \text{(S-HYP)}$$

$$\frac{\mathtt{class\ C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends\ C}'\ \{\ \overline{\mathtt{M}}\ \}}{\Gamma, \mathtt{x}:\mathtt{C}, \Delta \vdash \mathtt{x} :: \mathtt{C}'} \quad \text{(S-CLASS)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{S}, \Delta \vdash \mathtt{S} == \mathtt{T}}{\Gamma, \mathtt{x}:\mathtt{S}, \Delta \vdash \mathtt{x} :: \mathtt{T}} \quad \text{(S-REFL)}$$

$$\frac{\Gamma \vdash \mathtt{x} :: \mathtt{S} \qquad \mathtt{y}\text{ fresh} \qquad \Gamma, \mathtt{y}:\mathtt{S} \vdash \mathtt{y} :: \mathtt{T}}{\Gamma \vdash \mathtt{x} :: \mathtt{T}} \quad \text{(S-TRANS)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{S}, \mathtt{c}[\mathtt{x}/\mathtt{self}], \Delta \vdash \mathtt{x} :: \mathtt{T}}{\Gamma, \mathtt{x}:\mathtt{S}\{\mathtt{c}\}, \Delta \vdash \mathtt{x} :: \mathtt{T}} \quad \text{(S-CONST-L)}$$

$$\frac{\Gamma \vdash \mathtt{c}[\mathtt{x}/\mathtt{self}], \mathtt{x} :: \mathtt{T}}{\Gamma \vdash \mathtt{x} :: \mathtt{T}\{\mathtt{c}\}} \quad \text{(S-CONST-R)}$$

$$\frac{\Gamma, \mathtt{y}:\mathtt{R}, \mathtt{x}:\mathtt{S}, \Delta \vdash \mathtt{x} :: \mathtt{T}}{\Gamma, \mathtt{x}:\exists \mathtt{y}:\mathtt{R}.\ \mathtt{S}, \Delta \vdash \mathtt{x} :: \mathtt{T}} \quad \text{(S-EXISTS-L)}$$

$$\frac{\Gamma \vdash \mathtt{t}:\mathtt{R}, \mathtt{y} :: \mathtt{T}[\mathtt{t}/\mathtt{x}]}{\Gamma \vdash \mathtt{y} :: \exists \mathtt{x}:\mathtt{R}.\ \mathtt{T}} \quad \text{(S-EXISTS-R)}$$

Figure 7. Subtyping rules.

constraints, etc. It is therefore necessary to extract from the typing environment a context for the value constraint solver to reason about.

In Figure 5, we define the *constraint projection* $\sigma(\Gamma)$ that, in essence, strips out all type information from Γ, materializes field names and existentials, and drops subtyping constraints. In the last rule, we assume that alpha-equivalence is used to choose a variable `z` that does not occur in the context under construction. The dual projection π also defined in Figure 5 is discussed later.

If $\mathtt{c}_0$ is a value constraint, we specify that $\Gamma \vdash \mathtt{c}_0$ if $\sigma(\Gamma)$ entails $\sigma(\mathtt{c}_0)$ in $\mathcal{X}$ with rule X-PROJ in Figure 6.

5. Subtyping. We say that `S` is a subtype of `T` in Γ and write "$\Gamma \vdash \mathtt{S} <: \mathtt{T}$" if, informally, an expression of type `S` may be used when an expression of type `T` is required. The type-checking rules for method and constructor invocations for example make use of the subtyping relation.

Because of dependent types, both `S` and `T` may constrain `self`. Intuitively, `self` in `S` and `self` in `T` are to be thought as the same variable when evaluating the validity of the judgment "$\Gamma \vdash \mathtt{S} <: \mathtt{T}$". It is therefore necessary to instantiate `self`—equate `self` in both types with a fresh variable name—to reason about the subtyping relation. When formalizing subtyping and making proofs about it, we came to realize that this is cumbersome and error prone. This motivates the introduction of an alternate notation for subtyping judgments, which we now describe.

We adopt "$\Gamma \vdash \mathtt{x} :: \mathtt{T}$" as our primary form of subtyping judgment. If variable `x` is declared with type `S` in Γ, then this stands for "$\Gamma \vdash \mathtt{S}\{\mathtt{self} == \mathtt{x}\} <: \mathtt{T}\{\mathtt{self} == \mathtt{x}\}$". In other words, if `x` is fresh, the following equivalence holds:

$$\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T} \Leftrightarrow \Gamma \vdash \mathtt{S} <: \mathtt{T}$$

We use this equivalence to prove subtyping constraints in guards and class invariants (see rule S-SUB in Figure 7) but, as much as possible, we stick to the "::" form. In essence, it lets us introduce a variable name `x` to reason about once, which can be then used across an entire deduction tree.

The intent of FXG subtyping is to combine nominal subtyping and constraint entailment: type $\mathtt{C}\{\mathtt{c}\}$ is a subtype of $\mathtt{C}'\{\mathtt{c}'\}$ if `C` inherits from $\mathtt{C}'$ and `c` entails $\mathtt{c}'$. For example, the type "`RectArray{self.t==Int}`" is a subtype of "`Array{self.t<:Number}`" if `Int` is a subtype of `Number` and `RectArray` a subtype of `Array`.

The subtyping relation is specified in Figure 7. It relies of the constraint projection π of Figure 5 to extract subtyping constraints from the environment.

There are three sources of subtypes: (i) the `extends` relation of the source program (rule S-CLASS), (ii) subtyping constraints in the source program (rule S-HYP), and (iii) term equivalence (rule S-REFL). This last rule lets us for instance derive that "$\mathtt{x} : \mathtt{Type}\{\mathtt{self} == \mathtt{Int}\} \vdash \mathtt{x} :: \mathtt{Int}$." Subtyping is reflexive thanks to rule S-REFL and transitive by rule S-TRANS.

Rules S-CONST-L, S-CONST-R, S-EXISTS-L, and S-EXISTS-R handle constrained and existential types. In rule S-EXISTS-L, well-formedness ensures that variable `y` is not free in Δ or `T`.

Rules S-CONST-L and S-CONST-R let us rearrange constraints in types, e.g., $\mathtt{x}:\mathtt{T}\{\mathtt{c}, \mathtt{c}'\} \vdash \mathtt{x} :: \mathtt{T}\{\mathtt{c}\}\{\mathtt{c}'\}$.

6. Inconsistent subtyping constraints. We say that an environment Γ is *inconsistent* iff $\Gamma \vdash \mathtt{T} <: \mathtt{C}, \mathtt{T} <: \mathtt{D}$ for some type `T` and distinct class types `C` and `D` such that `C` does not inherit from `D` or vice versa.

In the remainder of the type system, that is, in member lookup and type-checking rules, we assume all environments

$$\frac{\texttt{class C}(\overline{\texttt{f}}:\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends C}'\ \{\ \overline{\texttt{M}}\ \}}{\Gamma \vdash \texttt{C.f}_i \Longrightarrow \texttt{C.f}_i:\texttt{T}_i} \quad \text{(H-FIELD)}$$

$$\frac{\texttt{class C}(\overline{\texttt{f}}:\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends C}'\ \{\ \overline{\texttt{M}}\ \} \qquad \texttt{def m}(\overline{\texttt{x}}:\overline{\texttt{T}'})\{\texttt{c}'\}:\texttt{R} = \texttt{e} \in \overline{\texttt{M}}}{\Gamma \vdash \texttt{C.m} \Longrightarrow \texttt{C.m}(\overline{\texttt{x}}:\overline{\texttt{T}'})\{\texttt{c}'\}:\texttt{R}} \quad \text{(H-METHOD)}$$

$$\frac{\Gamma \vdash \texttt{S} <: \texttt{T} \qquad \Gamma \vdash \texttt{T.i} \Longrightarrow \texttt{I}}{\Gamma \vdash \texttt{S.i} \Longrightarrow \texttt{I}} \quad \text{(H-SUB)}$$

$$\frac{\Gamma \vdash \texttt{T.i} \Longrightarrow \texttt{I} \qquad \forall \texttt{I}'.\ \Gamma \vdash \texttt{T.i} \Longrightarrow \texttt{I}' \Rightarrow \texttt{I} \ll \texttt{I}'}{\Gamma \vdash \texttt{T.i} \longrightarrow \texttt{I}} \quad \text{(H-AMB)}$$

$$\texttt{C.i}:\texttt{I} \ll \texttt{C.i}:\texttt{I} \quad \text{(O-REFL)}$$

$$\frac{\vdash \texttt{C} <: \texttt{C}' \qquad \texttt{this}:\texttt{C}, \overline{\texttt{x}}:\overline{\texttt{T}}, \texttt{c}' \vdash \texttt{c} \qquad \texttt{this}:\texttt{C}, \overline{\texttt{x}}:\overline{\texttt{T}}, \texttt{c} \vdash \texttt{R} <: \texttt{R}'}{\texttt{C.m}(\overline{\texttt{x}}:\overline{\texttt{T}})\{\texttt{c}\}:\texttt{R} \ll \texttt{C}'.\texttt{m}(\overline{\texttt{x}}:\overline{\texttt{T}})\{\texttt{c}'\}:\texttt{R}'} \quad \text{(O-METHOD)}$$

Figure 8. Member lookup. $\texttt{i}$ ranges over member names, $\texttt{I}$ over member signatures.

$$\Gamma, \texttt{x}:\texttt{T}, \Delta \vdash \texttt{x}:\texttt{T}\{\texttt{self} == \texttt{x}\} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \texttt{e}:\texttt{S}, \texttt{S.f} \longrightarrow \texttt{C.f}:\texttt{T} \qquad \mathsf{fields}(\texttt{C}) = \overline{\texttt{f}}:\overline{\texttt{T}} \qquad \texttt{x fresh}}{\Gamma \vdash \texttt{e.f}:\exists \texttt{x}:\texttt{S}.\ \texttt{T}[\texttt{x}.\overline{\texttt{f}}/\overline{\texttt{f}}]\{\texttt{self} == \texttt{x.f}\}} \quad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash \overline{\texttt{e}}:\overline{\texttt{S}} \qquad \mathsf{fields}(\texttt{C}) = \overline{\texttt{f}}:\overline{\texttt{T}} \qquad \overline{\texttt{x}}\ \text{fresh} \qquad \Gamma, \overline{\texttt{x}}:\overline{\texttt{S}} \vdash \overline{\texttt{x}} :: \overline{\texttt{T}}[\overline{\texttt{x}}/\overline{\texttt{f}}], \mathsf{inv}(\texttt{C})[\overline{\texttt{x}}/\texttt{this}.\overline{\texttt{f}}]}{\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}):\exists \overline{\texttt{x}}:\overline{\texttt{S}}.\ \texttt{C}\{\texttt{self} == \texttt{new C}(\overline{\texttt{x}})\}} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \texttt{e}:\texttt{S}, \overline{\texttt{e}}:\overline{\texttt{T}}, \texttt{S.m} \longrightarrow \texttt{C.m}(\overline{\texttt{x}}:\overline{\texttt{T}'})\{\texttt{c}\}:\texttt{R} \qquad \texttt{y}, \overline{\texttt{z}}\ \text{fresh} \qquad \theta = [\texttt{y}, \overline{\texttt{z}}/\texttt{this}, \overline{\texttt{x}}] \qquad \Gamma, \texttt{y}:\texttt{S}, \overline{\texttt{z}}:\overline{\texttt{T}} \vdash \texttt{c}\theta, \overline{\texttt{z}} :: \overline{\texttt{T}'\theta}}{\Gamma \vdash \texttt{e.m}(\overline{\texttt{e}}):\exists \texttt{y}:\texttt{S}, \overline{\texttt{z}}:\overline{\texttt{T}}.\ \texttt{R}\theta} \quad \text{(T-INVK)}$$

$$\frac{\Gamma \vdash \texttt{e}:\texttt{S}}{\Gamma \vdash \texttt{e as T}_\emptyset:\texttt{T}_\emptyset} \quad \text{(T-CAST)}$$

$$\Gamma \vdash \texttt{C}\{\texttt{c}\}:\texttt{Type}\{\texttt{self} == \texttt{C}\{\texttt{c}\}\} \quad \text{(T-CLASS)}$$

$$\frac{\begin{array}{c}\texttt{this}:\texttt{C}, \overline{\texttt{x}}:\overline{\texttt{T}}, \texttt{c} \vdash \texttt{e}:\texttt{S} \qquad \texttt{y}\ \text{fresh} \qquad \texttt{this}:\texttt{C}, \overline{\texttt{x}}:\overline{\texttt{T}}, \texttt{c}, \texttt{y}:\texttt{S} \vdash \texttt{y} :: \texttt{R} \\ \mathsf{method}(\mathsf{super}(\texttt{C}), \texttt{m}) = \texttt{m}(\overline{\texttt{x}'}:\overline{\texttt{T}'})\{\texttt{c}'\}:\texttt{R}' = \texttt{e}'\ \text{ implies }\ \texttt{C.m}(\overline{\texttt{x}}:\overline{\texttt{T}})\{\texttt{c}\}:\texttt{R} \ll \mathsf{super}(\texttt{C}).\texttt{m}(\overline{\texttt{x}'}:\overline{\texttt{T}'})\{\texttt{c}'\}:\texttt{R}'\end{array}}{\vdash \texttt{def m}(\overline{\texttt{x}}:\overline{\texttt{T}})\{\texttt{c}\}:\texttt{R} = \texttt{e}\ \text{OK in C}} \quad \text{(OK-METHOD)}$$

$$\frac{\texttt{this}:\texttt{C} \vdash \mathsf{inv}(\texttt{C}') \qquad \mathsf{fields}(\texttt{C}') = \overline{\texttt{f}'}:\overline{\texttt{T}'} \qquad \overline{\texttt{f}} \cap \overline{\texttt{f}'} = \emptyset \qquad \overline{\texttt{M}}\ \text{OK in C}}{\vdash \texttt{class C}(\overline{\texttt{f}}:\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends C}'\ \{\ \overline{\texttt{M}}\ \}\ \text{OK}} \quad \text{(OK-CLASS)}$$

Figure 9. Typing rules.

$\mathtt{x} : \mathtt{Type} \vdash \mathtt{x} : \mathtt{Type}\{\mathtt{self} == \mathtt{x}\}$ T-VAR

$\mathtt{x} : \mathtt{Type} \vdash \mathtt{new}\ \mathtt{C}(\mathtt{x}) : \mathtt{S}$ where $\mathtt{S}$ is $\exists \mathtt{z} : \mathtt{Type}\{\mathtt{self} == \mathtt{x}\}.\ \mathtt{C}\{\mathtt{self} == \mathtt{new}\ \mathtt{C}(\mathtt{z})\}$ T-NEW

$\mathtt{x} : \mathtt{Type} \vdash \mathtt{new}\ \mathtt{C}(\mathtt{x}).\mathtt{f} : \mathtt{T}$ where $\mathtt{T}$ is $\exists \mathtt{z} : \mathtt{S}.\ \mathtt{Type}\{\mathtt{self} == \mathtt{z}.\mathtt{f}\}$ T-FIELD

$\mathtt{x} : \mathtt{Type}, \mathtt{y} : \mathtt{T} \vdash \mathtt{y} :: \mathtt{Type}$ S-CONST-L and S-EXISTS-L

$\mathtt{x} : \mathtt{Type}, \mathtt{y} : \mathtt{T} \vdash \mathtt{y} == \mathtt{x}$ X-PROJ

Figure 10. Example judgments for program "`class C(f:Type) extends Object {}`."

are not inconsistent. In other words, inference rules cannot be instantiated with inconsistent environments.

This consistency check is key to the soundness proof. It ensures that if a method invocation is typed using the signature of method `m` in class `C` then, at run time, this invocation will be dispatched to `m` in either `C` or a subclass of `C` (as opposed to a class possibly unrelated to `C`).

While this criterion is adequate for FXG with its single-inheritance hierarchy, in the X10 type checker, we implement a refined consistency test which accounts for interfaces in addition to class inheritance.

7. Member lookup. Figure 8 specifies the field and method signatures available on each type.

A field signature is of the form "$\mathtt{C}.\mathtt{f} : \mathtt{T}$" with the name `C` of the class declaring the field, the field name `f` and the declared type for the field `T` (possibly a path type). Similarly, a method signature is written "$\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{R}$" with the class name `C`, method name `m`, formal names $\overline{\mathtt{x}}$ and types $\overline{\mathtt{T}}$, guard `c`, and return type `R`.

Lookup is a function of the receiver's type `T` and the desired member name `i`. In particular for methods it does not involve the formal types, argument types, or method guards—we do not consider overloading. The types and method guard will be checked later (see rule T-INVK in Figure 9).

We first define *ambiguous lookup*: member `i` of type `T` ambiguously resolves to signature `I` in context Γ, written "$\Gamma \vdash \mathtt{T}.\mathtt{i} \Longrightarrow \mathtt{I}$". Ambiguous lookup collects candidate signatures by looking at all the class types that are super types of `T`, which involves not only the inheritance tree but also the subtyping constraints in the input program.

Then, rule H-AMB, $\mathtt{T}.\mathtt{i}$ (unambiguously) resolves to `I`, written "$\Gamma \vdash \mathtt{T}.\mathtt{i} \longrightarrow \mathtt{I}$", iff $\mathtt{T}.\mathtt{i}$ resolves to `I` ambiguously and `I` overrides any other signature $\mathtt{T}.\mathtt{i}$ resolves to.

The overriding relation "$\ll$" is reflexive. Fields cannot be overridden in subclasses. A method of class `C` overrides a method of class $\mathtt{C}'$ it inherits from iff it has the same formal names and types, the guard of the method in $\mathtt{C}'$ entails the guard of the method in `C` and the return type in `C` is a subtype of the return type in $\mathtt{C}'$.

Of course, for the non-generic fragment of FXG, it would make sense to look for fields and methods by walking the class hierarchy bottom up, stopping at the first match, thus avoiding the need for ambiguous lookup altogether. But once there are type variables and bounds, it gets complicated. Moreover, there is not much point specifying an efficient traversal for a formal language like FXG without interfaces, so we stick to the inefficient but sound procedure of ambiguous lookup followed by ambiguity resolution.

8. Typing. The typing rules are specified in Figure 9. We write $\mathsf{inv}(\mathtt{C})$ for the invariant of class `C` and $\mathsf{super}(\mathtt{C})$ for the superclass of `C`.

T-VAR asserts the constraint "$\mathtt{self} == \mathtt{x}$," which records that any value of this type is known statically to be equal to `x`. Thanks to this constraint, we can for instance type check the invocation "`x.distance(x)`" on a variable `x` of type `Point` even if the `rank` of `x` in statically unknown.

T-FIELD resolves the field name on the expression type. Like T-VAR it records more than just the resolved field type `T`. It asserts that there exists an `x` of the receiver's type such that any value of this type is known statically to be equal to $\mathtt{x}.\mathtt{f}$. Because `T` may be a dependent type, we need to substitute possible occurrences of field names in `T` with the corresponding fields of `x`.

T-NEW has a similar structure to T-FIELD. It checks that the static types of the constructor arguments are subtypes of the declared field types and also imply the class invariant. Finally, it records that the types of the fields of the constructed object are the types of the constructor call arguments, which are typically more precise than (as in strict subtypes of) the declared field types of the class.

Combining these three rules with constraint entailment, we can for example in Figure 10 establish statically for program "`class C(f:Type) extends Object {}`" that if `x` is a type variable then "`new C(x).f`" has not only a type `T` that is a subtype of `Type` but in addition that any variable of type `T` is equal to `x`.

T-INVK similarly enforces that the argument types are subtypes of the types of the formals and checks that the method guard is entailed by the argument types.

T-CAST only requires `e` to be of some type `S`. At run time, the reduced value for `e` is checked to see if it is actually of type `T` (see R-CAST in Figure 3).

T-CLASS like T-VAR records that any value of this type is statically known to be $\mathtt{C}\{\mathtt{c}\}$.

OK-METHOD and OK-CLASS enforce overriding rules for fields and methods. The class invariant of a class must entail the class invariant of its superclass. OK-METHOD makes sure the body of a method has a type that is a subtype of the declared type (assuming the class invariant and the method guard).

3. Soundness

The following results hold for FXG irrespective of the choice of the value constraint system $\mathcal{X}$.

Lemma 3.1 (Principal types)**.** $\Gamma \vdash \mathtt{e} : \mathtt{S}$ *and* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *then* $\mathtt{S}$ *are* $\mathtt{T}$ *are identical.*

Lemma 3.2 (Progress)**.** *If* $\vdash \mathtt{e} : \mathtt{T}$ *then one of the following conditions holds:*

1. $\mathtt{e}$ *is a value,*
2. $\mathtt{e}$ *contains a stuck cast sub-expression, that is, an expression of the form "*$\mathtt{v}\ \mathtt{as}\ \mathtt{T}_{\mathtt{0}}$*,"*
3. *there exists* $\mathtt{e}'$ *such that* $\mathtt{e} \to \mathtt{e}'$.

Lemma 3.3 (Subject Reduction)**.** *If* $\mathtt{P}$ *is well typed and* $\mathtt{e} \to \mathtt{e}'$*, and* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *then there exists a type* $\mathtt{S}$ *such that* $\Gamma \vdash \mathtt{e}' : \mathtt{S}$*. Moreover,* $\Gamma \vdash \mathtt{S} <: \mathtt{T}$.

Theorem 3.4 (Type soundness)**.** *If* $\mathtt{P}$ *is well typed and* $\vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e}$ *reduces to a normal form* $\mathtt{e}'$ *then either* $\mathtt{e}'$ *contains a stuck cast sub-expression of the form "*$\mathtt{v}\ \mathtt{as}\ \mathtt{T}_{\mathtt{0}}$*" or* $\mathtt{e}'$ *is a value* $\mathtt{v}$ *and there exists* $\mathtt{S}$ *such that* $\vdash \mathtt{v} : \mathtt{S}$*. Moreover, in that case,* $\vdash \mathtt{S} <: \mathtt{T}$.

Constructors calls in a well-typed program do not violate class invariants at run time.

Theorem 3.5 (Class invariants)**.** *If* $\mathtt{P}$ *is well typed and* $\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{T}$ *and* $\overline{\mathtt{e}} \to^* \overline{\mathtt{v}}$ *then* $\Gamma \vdash \mathsf{inv}(\mathtt{c})[\overline{\mathtt{v}}/\mathtt{this}.\overline{\mathtt{f}}]$.

Method invocations in a well-typed program do not violate method guards at run time.

Theorem 3.6 (Method guards)**.** *If* $\mathtt{P}$ *is well typed and* $\Gamma \vdash \mathtt{e}.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{S}$ *and* $\mathtt{e} \to^* \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$ *and* $\overline{\mathtt{e}} \to^* \overline{\mathtt{w}}$ *and* $\mathsf{method}(\mathtt{C},\mathtt{m}) = \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{R} = \mathtt{e}'$ *then* $\Gamma \vdash \mathtt{c}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}), \overline{\mathtt{w}}/\mathtt{this}, \overline{\mathtt{x}}]$.

The proofs of these results are sketched in Appendix A.

4. Extensions

We now discuss possible extensions of FXG, first for the case of value constraints, then for type constraints.

Primitive types. Since the FXG design is parametric in the value constraint language we can easily extend it to support, say, arithmetic constraints or constraints on primitive types.

First, we assume we are given a constraint system $\mathcal{X}$ with a vocabulary of primitive types $\mathtt{R}$, functions $\mathtt{h}$, predicates $\mathtt{q}$, and literals $\mathtt{l}$ of these primitive types. Second, we extend the productions, operational semantics, and type system of FXG with the productions and inference rules of Figure 11. Formally, we should also extend the constraint projections, but the extensions are straightforward and omitted.

We denote $\mathsf{Rng}(\mathtt{l})$ the primitive type of the literal $\mathtt{l}$. We assume each function $\mathtt{h}$ is a total mapping from $\mathsf{Dom}(\mathtt{h})$ to $\mathsf{Rng}(\mathtt{h})$, that is, if $\vdash \overline{\mathtt{v}} : \mathsf{Dom}(\mathtt{h})$ then there exists a unique literal $\mathtt{l}$ equal to $\mathtt{h}(\overline{\mathtt{v}})$ and moreover $\mathsf{Rng}(\mathtt{l})$ is $\mathsf{Rng}(\mathtt{h})$.

For instance, if $\mathcal{X}$ defines the type `Int`, integer literals, the addition operator, and the greater-or-equal predicate, we could declare:

```
class Count(n:Int) extends Object {
  def inc():Count{self.n>=this.n} =
    new Count(this.n+1);
}
```

In rule T-FUN, we assume we are given an *abstraction* $\underline{\mathtt{h}}$ of every function $\mathtt{h}$. Formally, $\underline{\mathtt{h}}(\overline{\mathtt{x}})$ is a formula of the constraint language relating the variables $\overline{\mathtt{x}}$ and possibly `self`. For instance, the absolute value function "`abs`" could be typed as:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \mathtt{T} <: \mathtt{Int}}{\Gamma \vdash \mathtt{abs}(\mathtt{e}) : \exists \mathtt{x} : \mathtt{T}.\ \mathtt{Int}\{\mathtt{self} >= \mathtt{0}\}}$$

Informally, FXG+primitives is sound iff function abstractions are sound. Formally, we not only extended the constraint language but also the expression language, operational semantics, and type checking rules. Therefore, the soundness results of the previous section are not immediately applicable to FXG+primitives. But they are easily generalized because the proof structures are unchanged and need only be extended to the new rules. Principal types and progress hold unconditionally. Subject reduction depends on the function abstractions, which must be such that:

$$\begin{array}{rcl} \mathtt{h}(\overline{\mathtt{e}}) \to \mathtt{h}(\overline{\mathtt{e}}') \wedge \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}, \overline{\mathtt{e}}' : \overline{\mathtt{T}} & \Rightarrow & \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{S}}, \overline{\mathtt{y}} : \overline{\mathtt{T}}, \underline{\mathtt{h}}(\overline{\mathtt{y}}) \vdash \underline{\mathtt{h}}(\overline{\mathtt{x}}) \\ \mathtt{h}(\overline{\mathtt{v}}) \text{ evaluates to } \mathtt{l} & \Rightarrow & \mathtt{self} == \mathtt{l} \vdash \underline{\mathtt{h}}(\overline{\mathtt{v}}) \end{array}$$

Intuitively, abstractions must be such that they retain or increase precision with each execution step.

Structural subtyping constraints. We add the constraint "$\mathtt{T}_{\mathtt{0}}\ \mathtt{has}\ \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{R}$" which states that type $\mathtt{T}_{\mathtt{0}}$ has a method available with the given signature. We extend the type checking rules in Figure 12. Ambiguous lookup directly accounts for structural subtyping constraints by rule H-STRUCT. A class type $\mathtt{C}$ with method signature $\mathtt{C}.\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{R}$ satisfies the constraint $\mathtt{C}\ \mathtt{has}\ \mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{R}$ by rule X-STRUCT. By attaching the method signatures to `Object`, we ensure actual method declarations have precedence over ones known to exist by means of structural constraints. A side-effect of our formalization is that by combining rules H-STRUCT and X-STRUCT every method signature of the program also ends up being attached to `Object`. But this is fine since this "virtual" method is only visible for types below the class of declaration of the method so that the virtual method is always going to be overridden by the actual method declaration.

We can prove that soundness is preserved with this extension by simply updating the proof that run-time dispatch always returns a method compatible with the method signature selected during type checking. The rest of the soundness proof is unchanged.

Methodology. With these two extensions, one can get a feel for how to extend FXG with more constraints. On the one hand, richer value-dependency is obtained with the addition of matching type-checking rules and run-time steps.

$$
\begin{array}{rrcl}
\text{(Type)} & \mathtt{T_0} & ::= & \mathtt{R} \\
\text{(Expression)} & \mathtt{e} & ::= & \mathtt{h}(\overline{\mathtt{e}}) \mid \mathtt{l} \\
\text{(Constraint term)} & \mathtt{t} & ::= & \mathtt{l} \\
\text{(Value constraint)} & \mathtt{c_0} & ::= & \mathtt{q}(\overline{\mathtt{t}}) \\
\text{(Value)} & \mathtt{v} & ::= & \mathtt{l}
\end{array}
$$

$$\vdash \mathtt{l} : \mathsf{Rng}(\mathtt{l})\{\mathtt{self} == \mathtt{l}\} \qquad \text{(T-LIT)}$$

$$\frac{\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}, \overline{\mathtt{T}} <: \mathsf{Dom}(\mathtt{h})}{\Gamma \vdash \mathtt{h}(\overline{\mathtt{e}}) : \exists \overline{\mathtt{x}} : \overline{\mathtt{T}}.\ \mathsf{Rng}(\mathtt{h})\{\underline{\mathtt{h}}(\overline{\mathtt{x}})\}} \qquad \text{(T-FUN)}$$

$$\frac{\mathtt{h}(\overline{\mathtt{v}}) \text{ evaluates to } \mathtt{l}}{\mathtt{h}(\overline{\mathtt{v}}) \to \mathtt{l}} \qquad \text{(R-FUN)}$$

$$\frac{\mathtt{e}_i \to \mathtt{e}_i'}{\mathtt{h}(\mathtt{v}_1, \ldots, \mathtt{v}_{i-1}, \mathtt{e}_i, \ldots, \mathtt{e}_n) \to \mathtt{h}(\mathtt{v}_1, \ldots, \mathtt{v}_{i-1}, \mathtt{e}_i', \ldots, \mathtt{e}_n)} \qquad \text{(RC-FUN)}$$

Figure 11. FXG+primitive types.

$$\frac{\Gamma \vdash \mathtt{T_0}\ \mathtt{has}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{R}}{\Gamma \vdash \mathtt{T_0.m} \Longrightarrow \mathtt{Object.m}(\overline{\mathtt{x}}:\overline{\mathtt{T}})\{\mathtt{c}\} : \mathtt{R}} \qquad \text{(H-STRUCT)}$$

$$\frac{\mathtt{class\ C}(\overline{\mathtt{f}}:\overline{\mathtt{T}})\{\mathtt{c}\}\ \mathtt{extends\ C'}\ \{\ \overline{\mathtt{M}}\ \} \qquad \mathtt{def\ m}(\overline{\mathtt{x}}:\overline{\mathtt{T'}})\{\mathtt{c'}\} : \mathtt{R} = \mathtt{e} \in \overline{\mathtt{M}} \qquad \Gamma \vdash \mathtt{T_0} <: \mathtt{C}}{\Gamma \vdash \mathtt{T_0}\ \mathtt{has}\ \mathtt{m}(\overline{\mathtt{x}}:\overline{\mathtt{T'}})\{\mathtt{c'}\} : \mathtt{R}} \qquad \text{(X-STRUCT)}$$

Figure 12. FXG+structural subtyping constraints.

Because these are tightly coupled, subject reduction and progress proofs just need to be extended with new cases, with the bulk of the proof unchanged. On the other hand, richer type-dependency is obtained with the addition of subtyping rules and lookup rules. There, we must ensure that run-time dispatch is faithful to compile-time lookup of field and method signatures.

5. Towards a Practical Language

In this section, we discuss how the FXG formal system can be realized in a practical programming language. The choice of type variables and constraint system and other design factors affect the ease of use and ease of implementation of the resulting language. We outline the design and implementation choices made by X10, focusing on how FXG forms the core semantics of the language.

5.1 Type Variables

The first version of X10 did not support generic types. In extending the language, the first question to consider is the choice of type variables. Most object-oriented languages provide genericity by introducing *type parameters* on classes and methods. The development of a nominal OO type system with type parameters is now standard (cf. FGJ [20]). An alternative approach is to use *type members*, type-valued attributes of classes or objects. Virtual types in BETA [27] are an example of this approach, as are FXG's type-valued fields. Type members may be either statically bound to concrete types or dynamically bound at object creation time. Scala [38] supports both type parameters and type members.

Type parameters. Type parameters can be encoded as immutable type-valued fields in FXG. Unlike positional parameters, type fields can be referred to outside their class body—in constraints and in subclasses, for instance. Consequently, the encoding should rename type parameters to avoid name shadowing and ambiguity problems. In the following, we simply assume type fields are named to avoid conflicts. An an example, the Java class

```
class List<T> {
  void add(T x) { ... }
  void addAll(List<T> xs) { ... }
  T get(int i) { ... }
}
```

can be encoded as the following FXG class:

```
class List(T: Type) extends Object {
  def add(x: T) ...
  def addAll(xs: List{self.T==this.T}) ...
  def get(i: Int): T = ...
}
```

Instantiation of parameters is encoded as an equality constraint. A use of the type `List<C>` is encoded as the FXG type `List{T==C}`.

Parameter bounds can be encoded as subtyping constraints in either constrained kinds or in the class invariant. For example, the Java class

```
class Folder<T extends Foldable> { ... }
```

can be encoded as either of the following FXG classes:

```
class Folder(T: Type{self <: Foldable}) { ... }
class Folder(T: Type) {T <: Foldable} { ... }
```

A key issue with parametrized types affecting the expressiveness and usability of the language is *variance*: that is, what is the subtyping relationship between `C<S>` and `C<T>`? At present, type parameters in X10 are *invariant*—that is,

C<S> and C<T> are subtypes only if S and T are equal. Outlined below are several options for supporting variance in X10, building on the FXG formalism.

It is often expected that, for example, List<Int> is a subtype of List<Number> when Int is a subtype of Number. This *covariance*, however, is unsound if List<T> has methods that take T as an argument. As an example, if the T parameter of the List class above were covariant, then the following code would compile.

```
List<Number> nums = new List<Number>();
nums.add(new Float(2.718f));      // safe

List<Number> ints = new List<Int>();
ints.add(new Float(1.414f));      // unsafe
ints.addAll(nums);                // unsafe
```

Calling nums.add with a Float is safe since Float is a subtype of Number; however, calling ints.add with a Float is unsafe and can lead to a dynamic type error. Adding all elements of nums to ints will similarly fail. A sound type system should reject the above code.

Specification of variance can be done either at the *use site* or at the *definition site*. Java pioneered use-site variance through the use of *wildcard types* [46]. Scala and C♯ support definition-site variance annotations.

In *use-site variance*, the user of the generic type decides the variance of the type's parameters. In Java, variance is specified using bounded wildcard types. The type List<? extends Number> represents a list of some fixed, but statically unknown, element type that must be a subtype of Number. Both List<Int> and List<Number> are subtypes of this type. In contrast, there is no subtyping relationship between the *invariant* types List<Int> and List<Number>.

FXG can support use-site variance through subtyping constraints. The encoding of type parameters described above can be extended to handle wildcard types. For the above List class, the following correspondences hold:

Java	FXG
List<?>	List{true}
List<? extends C>	List{T<:C}
List<? super C>	List{T:>C}

Covariance relies on the result that if B is a subtype of A, then List{T<:B} is a subtype of List{T<:A}. However, like wildcard types [26], this encoding of covariance can hurt usability. Because programmers must make variance decisions for each use of a generic type, they must anticipate how that object will be used. In particular, if a type has a covariant constraint on a type variable, then methods that take that type variable as an argument cannot normally be called. For instance in the code below, the call to add is illegal:

```
val nums: List{T<:Number} = ...
nums.add(new Int(1));                // illegal
```

The problem is that nums.T is statically unknown. The compiler cannot determine if Int is a subtype of nums.T. Preventing the call ensures a dynamic type error does not occur. Since calls to methods like add that accept covariantly constrained parameters are illegal, objects with covariant type constraints can be rendered effectively read-only.

The other common approach to variance, *definition-site variance*, is used in Scala and C♯. In a class declaration, a parameter may be declared in-, co-, or contravariant. Following Kennedy and Pierce [23], variant parameters can be encoded in FXG using subtyping constraints at their use. All uses of Cons[A] are translated to Cons{T<:A}. If B is a subtype of A, then this encoding ensures the translation of Cons[B] is a subtype of the translation of Cons[A]—that is, Cons{T<:B} is a subtype of Cons{T<:A}. If T were an invariant parameter, the encoding of Cons[A] would be Cons{T==A}. For example, consider the following Scala declaration of a Cons cell with covariant parameter T.

```
class Cons[+T] {
  def head: T = ...
  def tail: Cons[T] = ...
}
```

This can be encoded in FXG as the class:

```
class Cons(T: Type) extends Object {
  def head: this.T = ...
  def tail: Cons{self.T<:this.T} = ...
}
```

To ensure type soundness in languages with definition-site variance, the use of variant parameter types in methods and fields must be restricted in the body of their class. The compiler checks that covariant type parameters do not occur in *negative* positions—that is, as method arguments—and that contravariant type parameters do not occur in *positive* positions—as method return types. These structural checks are needed to guarantee soundness, avoiding the dynamic type errors described above. In supporting definition-site variance, Scala does not permit, for example, an equivalent of the Java List<T> class above to be covariant in T. If T were covariant, then methods like add, which take a T as an argument would be prohibited. In the FXG encoding of definition-site variance as use-site constraints, these checks need not be performed, as long as the output of the encoding type-checks. The resulting FXG would not be able to invoke methods like add that lead to dynamic type errors.

Type members Rather than supporting genericity through type parameters, genericity could instead be provided with type members. Thorup [44] proposed using *virtual types* [14, 27, 28] to add genericity to Java. For example, a generic List class can be written as follows:

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

The virtual type T is unbound in List, but can be refined by binding T in a subclass:

```
class IntList extends List {
  final typedef T as Int;
}
```

Classes like List where the virtual type is not *final bound* to a concrete type must be abstract.

Virtual types, too, can be encoded as type-valued fields in FXG, similarly to how wildcards are encoded. In FXG, the analogous definition of the List class above is:

```
class List(T: Type) extends Object {
  def get(i: int): T { ... }
}
```

Bounds on virtual types can be encoded in the class invariant. For example, the subclass IntLit can constrain T to be equal to Int as follows:

```
class IntLit(){T==Int} extends List { }
```

However, unlike with virtual types. the FXG version of List need not be abstract; rather, T must be bound to a concrete type when an instance of List is created. Since immutable fields can be constrained where their class type is used (e.g., List{T<:Number} and List{T==Int}) a subclass of List need not be declared at all.

Since fields are inherited, the language design needs to account for ambiguities introduced when the same name is used for different fields declared in or inherited into a class. In FXG, a subclass cannot declare a field with the same name as one in a superclass; in a practical programming language, shadowing of field names could be allowed. Name conflicts can be disambiguated by "casting" the target to the desired supertype, e.g., (e as C).X specifies the field X inherited from C.

Because of these name ambiguity issues and because type parameters are more familiar to OO programmers, X10 chose to support type parameters rather than type members. Currently, type parameters in X10 are invariant. It is planned to extend the language with support for definition-site variance, basing the design on the FXG formalism, as outlined above.

5.2 Constraint System

The second design question is the choice of constraint system. Natural candidates are constraint systems that incorporate subtyping constraints or structural constraints on objects.

Subtyping constraints. The subtyping constraints in FXG can be incorporated into a full-fledged programming language like X10. For a type variable X one asserts the constraint X<:T. This constraint is realized by any valuation that maps X to a subtype of T. Constraints on types can specify either subtype (<:), supertype (:>), or equality bounds (==).

As described in the previous section, subtyping constraints in the class invariant provide a means to bound the type variables introduced by the class declaration. Constraints in constrained types C{c} can bound immutable type fields of the base type C. Subtyping constraints in method guards can bound type parameters of the method or bound type fields of the method's class. This feature is similar to optional methods in CLU [25] and to generalized type constraints in C♯ [13]. For instance, given a list of T, one could define a method print with a guard that requires that T be a subtype of Printable:

```
def print(){T <: Printable} {
  head.print();
  tail.print();
}
```

This constraint ensures that the head field of type T has a print() method.

Structural constraints. Rather than imposing nominal bounds on type variables, one can instead require that a type have a particular member—a field with a given name and type, or a method with a given name and signature. We introduce the constraints T has f:T and T has m($\overline{x}$:$\overline{S}$):T to express this. These constraints allow one to define an alternative version of the guarded print method above:

```
def print(){T has print(): Void} {
  head.print();
  tail.print();
}
```

With structural constraints, any list whose element type has a print method may be used, not just lists whose elements implement Printable.

Structural constraints on types are found in many languages. For instance, Haskell supports type classes [19, 22]. In Modula-3, type equivalence is structural rather than nominal as in object-oriented languages of the C family (e.g., C++, Java, and X10). Unity [30] is a Java-like language with both nominal and structural subtyping. Scala provides structural types as well.

In the class invariant, a structural constraint can bound the class's type variables, similar to the language PolyJ [35], which allows type parameters to be bounded using structural *where clauses* [10].

Because structural types are not supported directly on the Java virtual machine, implementing them on languages that target the JVM is non-trivial and can result in a performance penalty [11]. Structural constraints are not currently supported in X10, but are under consideration.

Default values. In languages like Java with primitive types, every type has a default value—null for reference types, false or 0 for primitive types. With constrained types, some types do not have an obvious default value. For example, the type C{self!=null} does not contain the value null.

Thus, a useful extension to the type system is to add constraints of the form `T haszero`. This constraint holds if the type `T` has a default value. Variables where the constraint does not hold must be explicitly initialized.

X10 supports default-value constraints in method guards. They are used primarily to enable construction of arrays of primitives or structs without providing an initial value for each array element. The default values are all represented by a 0 bit pattern, and array construction is implemented by requesting a zeroed out memory buffer.

5.3 Overloading and Dispatch

The next question to address is the overloading semantics for methods with constraints on formal parameters and with method guards. One option is to ignore constraints when checking for overloading. Thus, these three methods:

```
def m(List{T==Int,length==0}) = ...
def m(List{T==Int,length==n}) = ...
def m(List{T==Float,length==n}) = ...
```

are considered to have the same signature. It is a static error if more than one of these methods appears in the same class.

Alternatively, the overloading could be allowed, with methods resolved at compile-time using the constraint solver. It is an error if a call could resolve to more than one method. One question is whether to rule out overlapping methods (e.g., `m(Int{self>=0})` and `m(Int{self==1})`), or to permit them and have the caller resolve any ambiguities.

Going further, one could support a form of predicate dispatch [33], selecting the method to invoke by *dynamically* evaluating the constraints in the method signature and the method guard. With type constraints, multimethod dispatch could then be implemented as an extension of predicate dispatch.

X10 takes a conservative approach and does not allow overloading based on constraints or method guards.

5.4 Run-time Casts

While constraints are normally solved at compile time, constraints can be evaluated at run time by using casts. The expression `xs as List{length==n}` checks not only that `xs` is an instance of the `List` class, but also that `xs.length` equals `n`. An exception is thrown if the check fails.

The information needed to perform checked casts must be available at run time. Java's approach to generics implementation is to erase type parameters and to allow these casts with a static warning, but no dynamic check. Erasure admits more dynamic errors because it permits, for instance, a `C<A>` to be cast to `C<B>`. Retrieving a field of static type `B` could cause a run-time type error when an `A` is returned instead.

Unlike Java, X10 does not erase type parameters at run time. Instead, each instance of a generic type contains a description of the types that its parameters are instantiated upon. This extra run-time type information enables checked casts to generic types.

In the above example, the test of the constraint does not require run-time constraint solving; the constraint can be checked by simply evaluating the `length` field of `xs` and comparing against `n`. However, the situation is more complicated when casting to a generic type.

Similarly, the cast `xs as List{T<:C{c}}` checks that the element type of `xs` is a subtype of `C{c}`. This test requires a run-time constraint entailment test. Suppose `xs` were declared to be a `List{T==C{d}}`. Checking the above cast requires testing that `C{d}` is a subtype of `C{c}`. This check, in turn, requires checking that `d` entails `c`.

One approach is to restrict the language to rule out casts to type parameters and to generic types with subtyping constraints, ensuring that entailment checks are not needed at run time. Alternatively, the constraint solver could be embedded into the runtime system. However, this solution can result in inefficient run-time casts if entailment checking for the given constraint system is expensive.

The X10 implementation makes a compromise. Run-time type information is preserved, but constraints are not.

5.5 Static vs. Dynamic Checking

Checking constraints statically rather than at run time enables early error detection and allows the compiler to generate better code. However, during development, ensuring constraints hold at each compile can slow progress. These tradeoffs are similar to the tradeoffs between static and dynamic typing. The X10 compiler supports two modes. In one mode, the compiler will reject programs when a constraint entailment cannot be proved; in another mode, similar to Flanagan's hybrid typing [15], the compiler emits dynamic checks for these entailments. Dynamic checks need to be performed to check class invariants when new objects are created, to check method guards, and to check assignments from subtypes to supertypes if the solver cannot determine that the assignment is allowed. Emitting dynamic checks can also permit a more expressive constraint language, allowing programmers to write constraints that cannot (yet) be handled by the embedded solver.

5.6 Inconsistent Constraints

The soundness of the type system ensures that constraints cannot be violated at run time. If a class invariant or a constraint on a type is inconsistent, then no values of that type can exist at run time. Similarly, if a method guard is inconsistent, that method cannot be called. Any code dependent on an inconsistent guard is unreachable.

For subject reduction to hold, the formal system assumes that subtyping constraints are not inconsistent; however, other constraints may be. The compiler can therefore allow inconsistent constraints. For developers, it is useful for the compiler to report whether a constraint is inconsistent. However, this requires the constraint system to be complete. Hence, the X10 compiler is more strict about type constraints than about value constraints. The compiler enforces consis-

tency of constraints on types, but not constraints on values. In practice, this means the X10 compiler accepts the following method, even though it can never be invoked:

```
def m(x: Int){x==0, x==1} ...
```

But it rejects the analogous method with type parameters rather than value parameters:

```
def p(X: Type){X==C, X==D} ...
```

where `C` and `D` are classes.

5.7 Mutable State

Objects in FXG contain only immutable value and type fields. X10, additionally, supports mutable and immutable instance fields. Constraints continue to be invariants on only the immutable state of objects (including types). Allowing constraints on mutable data would not be sound since a constraint that holds at one point in the program might not hold at another.

One subtlety is ensuring that class invariants are established correctly. When a constructor executes, fields of the receiver are initialized one-by-one, which can potentially allow the object being constructed to be accessed before the class invariant is established for the object. To address this, X10 distinguishes between fields and *properties*. Properties are immutable (final) fields of the object. Unlike normal fields, X10 requires that all properties of the object be initialized instantaneously. This provides a single program point—a `property` statement—at which the compiler can check if the class invariant holds. Before this point, the properties of the object cannot be accessed; after this point, the class invariant is established.

Unlike properties, final fields need not be initialized all at once. As in Java, final fields can be initialized at any point during constructor execution. However, fields cannot be used in constraints.

6. Related Work

Constraint-based type systems, dependent types, and generic types have been well studied in the literature. Further discussion of related work for constrained types can be found in our earlier work [37].

Constraint-based type systems. The use of type constraints for type inference and subtyping was first proposed by Mitchell [34] and Reynolds [39]. HM(X) [41] is a constraint-based framework for Hindley–Milner-style type systems. The framework is parametrized on the specific constraint system X; instantiating X yields extensions of the HM type system. Constraints in HM(X) are over types, not values. The HM(X) approach is an important precursor to our constrained types approach. The principal difference is that HM(X) applies to functional languages and does not integrate dependent types. We consider object-oriented languages with constraint-based type systems when we discuss generic types, below.

Dependent types. Dependent type systems [3, 32, 49] parametrize types on values. Our work is closely related to Dependent ML (DML [49]), which is also built parametrically on a constraint solver. The main distinction between DML and constrained types lies in the target domain: DML is a functional programming language; constrained types are designed for imperative, concurrent object-oriented languages. Types in DML are refinement types [16]: they do not affect the operational semantics, and erasing the constraints yields a legal DML program. This differs from generic constrained types, where erasure of subtyping constraints can prevent the program from type-checking. DML does not permit any run-time checking of constraints (dynamic casts). Another distinction between DML and constrained types is that constraints in DML are defined over a set of "index" variables; in X10, constraints are defined over program variables and types.

Liquid types [40], 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 in the relation and if one type's qualifier implies the other's. Liquid types support type inference and the type system is path sensitive; neither is the case in X10. Liquid types do not provide subtyping constraints.

Bierman et al. [4] propose a functional language with refinement types. Rather than use the constraint solver as a subroutine for subtyping checks, type-checking is performed by an SMT solver by translating types into logical formulas. The language supports a richer set of predicates on values than X10, but this is in large part orthogonal to the rest of the language design. Their language does not include constraints on types.

Köksal et al. [24] takes another approach to integrating constraints with the type system. Logical variables are added to Scala, and an SMT solver is used to solve constraints. Like Bierman et al. [4], any pure function can be used in a constraint.

Genericity. Genericity in object-oriented languages is usually supported through type parametrization.

A number of proposals for adding genericity to Java quickly followed the initial release of the language [1, 5, 35, 44]. GJ [5] implements invariant type parameters via type erasure. Java 5 [18] adopted the same implementation approach, incorporating wildcards and raw types [46]. Other proposals [8, 35, 47, 48] support run-time representation of type parameters. PolyJ [35] permits instantiation of parameters on primitive types and structural parameter bounds. MixGen [1] supported mixins through type parametrization.

Variance in Java is handled at the use-site using wildcards [7, 46]. Scala [38] and C♯ [12], by contrast, support definition-site variance annotations, which address many of the usability concerns of wildcards [26], but can often result in complicated or duplicated code to create invariant,

covariant, and contravariant versions of a library class. Altidor et al. [2] propose a framework for combining definition- and use-site variance in a Java-like language. Encoding this framework in FXG is an interesting area for future work.

Summers and Cameron et al. [6, 43] characterized wildcards in terms of existential types. Our encoding of wildcards in FXG similarly uses existentials, over constraint terms rather than types, however. Summers et al. [42, 43] observe that care must be taken to model assignment to avoid an unsoundness. We leave this extension for future work.

Virtual classes and *virtual types* [14, 27, 28] are another mechanism for supporting genericity, using nested types rather than parametrization. As discussed in Section 5.1, Thorup [44] proposed using virtual types to provide genericity in Java. Much of the development of Java's generics followed from virtual classes: use-site variance based on structural virtual types was proposed by Thorup and Torgersen [45] and extended for parametrized type systems by Igarashi and Viroli [21]; the latter type system lead to the development of wildcards in Java [7, 46]. Dependent classes [17] generalize virtual classes to express similar semantics via parametrization rather than nesting. With type properties, classes are not parametrized on their values; rather properties are members and types are constructed by constraining these properties. Parametrization can be encoded with type properties using equality constraints.

7. Conclusions

We have presented a constraint-based framework FXG for type- and value-dependent types in an object-oriented language. The use of constraints on type properties provides a framework for capturing many features of generics in object-oriented languages and then extending these features with more expressive power. We have proved the type system sound.

The type system of FXG formalizes the semantics of the X10 programming language. The design admits an efficient implementation for generics and dependent types in X10, available at `x10-lang.org`. To improve the expressiveness of X10, we plan to implement a type inference algorithm that infers constraints over types and values, and to support user-defined constraints.

Acknowledgments

This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proc. OOPSLA '03*, pages 96–114, October 2003.

[2] John Altidor, Shan Shan Huang, and Yannis Smaragdakis. Taming the wildcards: Combining definition- and use-site variance. In *Proc. 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2011.

[3] Lennart Augustsson. Cayenne: a language with dependent types. In *ACM SIGPLAN International Conf. on Functional Programming (ICFP '98)*, pages 239–250, 1998.

[4] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. *The Journal of Functional Programming*, 22(1):31–105, March 2012.

[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 *Proc. OOPSLA '98*, 1998.

[6] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *Formal Techniques for Java-Like Programs (FTfJP)*, July 2009.

[7] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *Proc. ECOOP '08*, number 5142 in Lecture Notes in Computer Science, pages 2–26, July 2008.

[8] 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.

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

[10] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proc. OOPSLA '95*, pages 156–168, Austin TX, October 1995. ACM SIGPLAN Notices 30(10).

[11] Gilles Dubochet and Martin Odersky. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. In *ICOOOLPS'09*, pages 34–41, 2009.

[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 *Proc. ECOOP '06*, 2006.

[14] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *33th ACM Symp. on Principles of Programming Languages (POPL)*, pages 270–282, Charleston, South Carolina, January 2006.

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

[16] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. 1991 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 268–277, June 1991.

[17] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *Proc. OOPSLA '07*, pages 133–152, 2007.

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

[19] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[20] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[21] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(5):795–847, 2006.

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

[23] Andrew Kennedy and Benjamin Pierce. On decidability of nominal subtyping with variance. In *Foundations of Object-Oriented Languages (FOOL)*, 2007.

[24] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *39th ACM Symp. on Principles of Programming Languages (POPL)*, pages 151–164, January 2012.

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

[26] Howard Lovatt. Simplyfing Java generics by eliminating wildcards, January 2008. Retrieved March 22, 2009.

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

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

[29] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Third Annual Symposium on Logic in Computer Science*, 1988.

[30] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *Proc. ECOOP '08*, number 5142 in Lecture Notes in Computer Science, July 2008.

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

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

[33] Todd Millstein. Practical predicate dispatch. In *Proc. OOPSLA '04*, October 2004.

[34] John C. Mitchell. Coercion and type inference. In *11th Annual ACM Symposium on Principles of Programming Languages (POPL'84)*, pages 174–185, 1984.

[35] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

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

[37] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proc. OOPSLA '08*, October 2008.

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

[39] John C. Reynolds. Three approaches to type structure. In *TAPSOFT/CAAP 1985*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, 1985.

[40] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proc. 2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2008.

[41] 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.

[42] Alexander J. Summers. Modelling Java requires state. In *Formal Techniques for Java-Like Programs (FTfJP)*, July 2009.

[43] Alexander J. Summers, Nicholas Cameron, Mariangiola Dezani-Ciancaglini, and Sophia Drossopoulou. Towards a semantic model for java wildcards. In *Formal Techniques for Java-Like Programs (FTfJP)*, June 2010.

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

[45] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining virtual types and parameterized classes. In *Proc. ECOOP '98*, 1998.

[46] 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.

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

[48] 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.

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

A. Proof Sketch

We assume well-formedness and non-inconsistent environments.

Lemma A.1. *If* $\Gamma \vdash \mathtt{T.i} \longrightarrow \mathtt{I}$ *and* $\Gamma \vdash \mathtt{T.i} \longrightarrow \mathtt{I}'$ *then* $\mathtt{I} = \mathtt{I}'$.

Proof. By H-AMB $\mathtt{I} \ll \mathtt{I}'$ and $\mathtt{I}' \ll \mathtt{I}$. If $\mathtt{I}$ is a field signature then by O-REFL $\mathtt{I} = \mathtt{I}'$. If $\mathtt{I}$ is a method signature then $\mathtt{I}'$ must be a method signature. Let $\mathtt{C}$ be the class of $\mathtt{I}$ and $\mathtt{C}'$ be the class of $\mathtt{I}'$. Suppose $\mathtt{C}$ is not $\mathtt{C}'$ then by O-METHOD $\mathtt{C} <: \mathtt{C}'$ and $\mathtt{C}' <: \mathtt{C}$. Contradiction. $\mathtt{C}$ has at most one method named $\mathtt{m}$, therefore $\mathtt{I} = \mathtt{I}'$ in all cases. □

Theorem A.2 (Principal Types)**.** *If* $\Gamma \vdash \mathtt{e} : \mathtt{S}$ *and* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *then* $\mathtt{S} = \mathtt{T}$.

Proof. By induction on the structure of $\mathtt{e}$. There is exactly one typing rule for each kind of expression. Moreover, by Lemma A.1, each field name or method name may resolve to at most one signature on a given type. Therefore, there is only one way the T-FIELD and T-INVK rules can be used to type a field selection or a method invocation. □

Theorem A.3 (Progress)**.** *If* $\vdash \mathtt{e} : \mathtt{T}$ *then one of the following conditions holds:*

1. $\mathtt{e}$ *is a value,*
2. $\mathtt{e}$ *has a stuck cast sub-expression of the form* $\mathtt{v}\ \mathtt{as}\ \mathtt{T}_0$*,*
3. *there exists* $\mathtt{e}'$ *such that* $\mathtt{e} \to \mathtt{e}'$*.*

Proof. By induction on the structure of the expression. Assume $\mathtt{e}$ contains no stuck cast sub-expression of the form $\mathtt{v}\ \mathtt{as}\ \mathtt{T}_0$ and is not a value.

- If $\mathtt{e}$ is $\mathtt{a.f}$.
  If $\mathtt{a}$ is a value then $\mathtt{e}$ can make a step by rule R-FIELD. Otherwise, by the induction hypothesis, $\mathtt{a} \to \mathtt{a}'$ then $\mathtt{e}$ can make a step by rule RC-FIELD.
- If $\mathtt{e}$ is $\mathtt{a.m}(\overline{\mathtt{b}})$.
  If $\mathtt{a}, \overline{\mathtt{b}}$ are values then $\mathtt{e}$ can make a step by rule R-INVK. Otherwise, if $\mathtt{a}$ is not a value then by the induction hypothesis $\mathtt{a} \to \mathtt{a}'$ and $\mathtt{e}$ can make a step by rule RC-INVK-RECV. Otherwise, if $\mathtt{b}_i$ is not a value then by the induction hypothesis $\mathtt{b}_i \to \mathtt{b}_i'$ and $\mathtt{e}$ can make a step by rule RC-INVK-ARG.
- If $\mathtt{e}$ is $\mathtt{new\ C}(\overline{\mathtt{a}})$
  Since $\mathtt{a}_i$ is not a value for some $\mathtt{i}$ then $\mathtt{e}$ can make a step by rule RC-NEW-ARG.
- If $\mathtt{e}$ is $\mathtt{a\ as}\ \mathtt{T}_0$.
  If $\mathtt{a}$ is not a value then $\mathtt{a}$ is well typed by T-CAST, hence can make a step by the induction hypothesis, thus $\mathtt{e}$ can make a step by rule RC-CAST. Otherwise, if $\mathtt{a}$ is a value then $\mathtt{e}$ can make a step by rule R-CAST since $\mathtt{e}$ contains no stuck cast sub-expression. □

Lemma A.4. *If* $\mathtt{P}$ *is well typed and* $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Gamma \vdash \mathtt{T.i} \longrightarrow \mathtt{I}$ *then there exists* $\mathtt{I}'$ *such that* $\Gamma \vdash \mathtt{S.i} \longrightarrow \mathtt{I}'$ *and* $\mathtt{I}' \ll \mathtt{I}$*.*

Proof. Let $\mathtt{C}$ be the class of $\mathtt{I}$. By H-SUB, $\Gamma \vdash \mathtt{S.i} \Longrightarrow \mathtt{I}$. By definition of ambiguous lookup, $\Gamma \vdash \mathtt{T} <: \mathtt{C}$. By S-TRANS, $\Gamma \vdash \mathtt{S} <: \mathtt{C}$. Let $\mathtt{I}'$ be such that $\Gamma \vdash \mathtt{S.i} \Longrightarrow \mathtt{I}'$ and $\mathtt{C}'$ the class of $\mathtt{I}'$. By definition of ambiguous lookup, $\Gamma \vdash \mathtt{S} <: \mathtt{C}'$. Because Γ is consistent, all such $\mathtt{C}'$ are related via inheritance. Let $\mathtt{C}''$ be the maximum of this set of classes and $\mathtt{I}''$ the corresponding signature. By OK-METHOD, $\mathtt{I}'' \ll \mathtt{I}'$ for all $\mathtt{I}'$ including $\mathtt{I}'' \ll \mathtt{I}$. By H-AMB, $\Gamma \vdash \mathtt{S.i} \longrightarrow \mathtt{I}''$. □

Lemma A.5. *If* $\mathtt{method}(\mathtt{C}, \mathtt{m}) = \mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M} = \mathtt{e}$ *then* $\Gamma \vdash \mathtt{C.m} \longrightarrow \mathtt{C}'.\mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M}$ *for* $\mathtt{C}'$ *a superclass of* $\mathtt{C}$ *or* $\mathtt{C}$*.*

Proof. Since Γ is consistent the only class types that $\mathtt{C}$ is a subtype of are $\mathtt{C}$ and the superclasses of $\mathtt{C}$. Let $\mathtt{C}'$ be $\mathtt{C}$ if $\mathtt{C}$ declares $\mathtt{m}$ or its lowest superclass that declared $\mathtt{m}$. By rule OK-METHOD $\mathtt{C}'.\mathtt{m}$ overrides all the methods $\mathtt{m}$ defined in these classes. Therefore, $\Gamma \vdash \mathtt{C.m} \longrightarrow \mathtt{C}'.\mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M}$. □

The following lemmas permit replacing one type by a subtype in various contexts.

Lemma A.6. *If* $\Gamma, \mathtt{x} : \mathtt{X} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Gamma \vdash \mathtt{e}' : \mathtt{Y}$ *and* $\Gamma, \mathtt{x} : \mathtt{Y} \vdash \mathtt{x} :: \mathtt{X}$ *then there exists* $\mathtt{S}$ *such that* $\Gamma \vdash \mathtt{e}[\mathtt{e}'/\mathtt{x}] : \mathtt{S}$ *and* $\Gamma, \mathtt{y} : \mathtt{S} \vdash \mathtt{y} :: \exists \mathtt{x} : \mathtt{Y}.\mathtt{T}$*.*

Lemma A.7. *If* $\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{c}$ *and* $\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T}$ *then* $\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{c}$*.*

Lemma A.8. *If* $\Gamma, \mathtt{y} : \mathtt{S} \vdash \mathtt{y} :: \mathtt{T}$ *then* $\Gamma, \mathtt{x} : \exists \mathtt{y} : \mathtt{S}.\mathtt{U} \vdash \mathtt{x} :: \exists \mathtt{y} : \mathtt{T}.\mathtt{U}$*.*

Lemma A.9. *If* $\Gamma, \mathtt{y} : \mathtt{U}, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T}$ *then* $\Gamma, \mathtt{x} : \exists \mathtt{y} : \mathtt{U}.\mathtt{S} \vdash \mathtt{x} :: \exists \mathtt{y} : \mathtt{U}.\mathtt{T}$*.*

Proof. Straightforward inductions. □

Theorem A.10 (Subject Reduction)**.** *If* $\mathtt{P}$ *is well typed and* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \to \mathtt{e}'$ *then there exists* $\mathtt{S}$ *such that* $\Gamma \vdash \mathtt{e}' : \mathtt{S}$*. Moreover* $\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T}$*.*

Proof. By induction on the proof of $\mathtt{e} \to \mathtt{e}'$. Assume $\Gamma \vdash \mathtt{e} : \mathtt{T}$. For simplicity, we omit substitutions from the proof. In other words, we do as if field lookup, method lookup, and the fields, method, and inv predicates return artefacts that are already matching our choice of fresh variables.

- $\mathtt{e.f} \to \mathtt{e}'.\mathtt{f}$ by rule RC-FIELD

T-FIELD	$\Gamma \vdash \mathtt{e} : \mathtt{R}$
	$\Gamma \vdash \mathtt{R.f} \longrightarrow \mathtt{C.f} : \mathtt{F}$
	$\Gamma \vdash \mathtt{e.f} : \mathtt{T}$
where	$\mathtt{T}$ is $\exists \mathtt{r} : \mathtt{R}.\mathtt{F}\{\mathtt{self} == \mathtt{r.f}\}$
Ind. hyp.	$\Gamma \vdash \mathtt{e}' : \mathtt{R}'$ and $\Gamma, \mathtt{x} : \mathtt{R}' \vdash \mathtt{x} :: \mathtt{R}$
Lemma A.4	$\Gamma \vdash \mathtt{R}'.\mathtt{f} \longrightarrow \mathtt{C.f} : \mathtt{F}$
T-FIELD	$\Gamma \vdash \mathtt{e}'.\mathtt{f} : \mathtt{S}$
where	$\mathtt{S}$ is $\exists \mathtt{r} : \mathtt{R}'.\mathtt{F}\{\mathtt{self} == \mathtt{r.f}\}$
Lemma A.8	$\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T}$

- $\mathtt{e.m}(\overline{\mathtt{a}}) \to \mathtt{e}'.\mathtt{m}(\overline{\mathtt{a}})$ by rule RC-INVK-RECV

T-INVK	$\Gamma \vdash \mathtt{e} : \mathtt{R}, \overline{\mathtt{a}} : \overline{\mathtt{A}}$
	$\Gamma \vdash \mathtt{R.m} \longrightarrow \mathtt{C.m}(\overline{\mathtt{x}} : \overline{\mathtt{X}})\{\mathtt{c}\} : \mathtt{M}$
	$\Gamma, \mathtt{r} : \mathtt{R}, \overline{\mathtt{x}} : \overline{\mathtt{A}} \vdash \mathtt{c}, \overline{\mathtt{x}} :: \overline{\mathtt{X}}$
	$\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{a}}) : \mathtt{T}$
where	$\mathtt{T}$ is $\exists \mathtt{r} : \mathtt{R}.\exists \overline{\mathtt{x}} : \overline{\mathtt{A}}.\mathtt{M}$
Ind. hyp.	$\Gamma \vdash \mathtt{e}' : \mathtt{R}'$ and $\Gamma, \mathtt{x} : \mathtt{R}' \vdash \mathtt{x} :: \mathtt{R}$
Lemma A.4	$\Gamma \vdash \mathtt{R}'.\mathtt{m} \longrightarrow \mathtt{C}'.\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{X}})\{\mathtt{c}'\} : \mathtt{M}'$
	$\Gamma, \mathtt{r} : \mathtt{R}', \overline{\mathtt{x}} : \overline{\mathtt{X}}, \mathtt{c}', \mathtt{y} : \mathtt{M}' \vdash \mathtt{y} :: \mathtt{M}$
	$\Gamma, \mathtt{r} : \mathtt{R}', \overline{\mathtt{x}} : \overline{\mathtt{X}}, \mathtt{c} \vdash \mathtt{c}'$
Lemma A.7	$\Gamma, \mathtt{r} : \mathtt{R}', \overline{\mathtt{x}} : \overline{\mathtt{A}} \vdash \mathtt{c}'$
	$\Gamma, \mathtt{r} : \mathtt{R}', \overline{\mathtt{x}} : \overline{\mathtt{A}}, \mathtt{y} : \mathtt{M}' \vdash \mathtt{y} :: \mathtt{M}$
T-INVK	$\Gamma \vdash \mathtt{e}'.\mathtt{m}(\overline{\mathtt{a}}) : \mathtt{S}$
where	$\mathtt{S}$ is $\exists \mathtt{r} : \mathtt{R}'.\exists \overline{\mathtt{x}} : \overline{\mathtt{A}}.\mathtt{M}'$
Lemma A.9	$\Gamma, \mathtt{y} : \mathtt{S} \vdash \mathtt{y} :: \exists \mathtt{r} : \mathtt{R}'.\exists \overline{\mathtt{x}} : \overline{\mathtt{A}}.\mathtt{M}$
Lemma A.8	$\Gamma, \mathtt{y} : \exists \mathtt{r} : \mathtt{R}'.\exists \overline{\mathtt{x}} : \overline{\mathtt{A}}.\mathtt{M} \vdash \mathtt{y} :: \mathtt{T}$
S-TRANS	$\Gamma, \mathtt{y} : \mathtt{S} \vdash \mathtt{y} :: \mathtt{T}$

- $\mathtt{v.m}(\overline{\mathtt{a}}) \to \mathtt{v.m}(\overline{\mathtt{a}}')$ by rule RC-INVK-ARG

T-INVK	$\Gamma \vdash \mathtt{v} : \mathtt{R}, \overline{\mathtt{a}} : \overline{\mathtt{A}}$
	$\Gamma \vdash \mathtt{R.m} \longrightarrow \mathtt{C.m}(\overline{\mathtt{x}} : \overline{\mathtt{X}})\{\mathtt{c}\} : \mathtt{M}$
	$\Gamma, \mathtt{r} : \mathtt{R}, \overline{\mathtt{x}} : \overline{\mathtt{A}} \vdash \mathtt{c}, \overline{\mathtt{x}} :: \overline{\mathtt{X}}$
	$\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{a}}) : \mathtt{T}$
where	$\mathtt{T}$ is $\exists \mathtt{r} : \mathtt{R}. \exists \overline{\mathtt{x}} : \overline{\mathtt{A}}. \mathtt{M}$
Ind. hyp.	$\Gamma \vdash \overline{\mathtt{a}'} : \overline{\mathtt{A}'}$ and $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{A}'} \vdash \overline{\mathtt{x}} :: \overline{\mathtt{A}}$
S-TRANS	$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{A}'} \vdash \overline{\mathtt{x}} :: \overline{\mathtt{X}}$
Lemma A.7	$\Gamma, \mathtt{r} : \mathtt{R}, \overline{\mathtt{x}} : \overline{\mathtt{A}'} \vdash \mathtt{c}$
T-INVK	$\Gamma \vdash \mathtt{v.m}(\overline{\mathtt{a}'}) : \mathtt{S}$
where	$\mathtt{S}$ is $\exists \mathtt{r} : \mathtt{R}. \exists \overline{\mathtt{x}} : \overline{\mathtt{A}'}. \mathtt{M}$
Lemma A.8	$\Gamma, \mathtt{y} : \mathtt{S} \vdash \mathtt{y} :: \mathtt{T}$

- $\mathtt{new\ C}(\overline{\mathtt{a}}) \to \mathtt{new\ C}(\overline{\mathtt{a}'})$ by rule RC-NEW-ARG

T-NEW	$\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{R}}$
	$\mathsf{fields}(\mathtt{C}) = \overline{\mathtt{f}} : \overline{\mathtt{F}}$
	$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{R}} \vdash \overline{\mathtt{x}} :: \overline{\mathtt{F}}, \mathsf{inv}(\mathtt{C})$
	$\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{T}$
where	$\mathtt{T}$ is $\exists \overline{\mathtt{x}} : \overline{\mathtt{R}}. \mathtt{C}\{\mathtt{self} == \mathtt{new\ C}(\overline{\mathtt{x}})\}$
Ind. hyp.	$\Gamma \vdash \overline{\mathtt{e}'} : \overline{\mathtt{R}'}$ and $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{R}'} \vdash \overline{\mathtt{x}} :: \overline{\mathtt{R}}$
S-TRANS	$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{R}'} \vdash \overline{\mathtt{x}} :: \overline{\mathtt{F}}$
Lemma A.7	$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{R}'} \vdash \mathsf{inv}(\mathtt{C})$
T-NEW	$\mathtt{new\ C}(\overline{\mathtt{e}'}) : \mathtt{S}$
where	$\mathtt{S}$ is $\exists \overline{\mathtt{x}} : \overline{\mathtt{R}'}. \mathtt{C}\{\mathtt{self} == \mathtt{new\ C}(\overline{\mathtt{x}})\}$
Lemma A.8	$\Gamma, \mathtt{y} : \mathtt{S} \vdash \mathtt{y} :: \mathtt{T}$

- $\mathtt{e\ as\ T} \to \mathtt{e}'\ \mathtt{as\ T}$ by rule RC-CAST

T-CAST	$\Gamma \vdash \mathtt{e\ as\ T} : \mathtt{T}$
	$\Gamma \vdash \mathtt{e} : \mathtt{S}$
Ind. hyp.	$\Gamma \vdash \mathtt{e}' : \mathtt{S}'$
T-CAST	$\Gamma \vdash \mathtt{e}'\ \mathtt{as\ T} : \mathtt{T}$
S-REFL	$\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} :: \mathtt{T}$

- $\mathtt{new\ C}(\overline{\mathtt{v}})\ \mathtt{as\ T} \to \mathtt{new\ C}(\overline{\mathtt{v}})$ by rule R-CAST

T-CAST	$\vdash \mathtt{new\ C}(\overline{\mathtt{v}}) : \mathtt{S}$
T-NEW	$\vdash \overline{\mathtt{v}} : \overline{\mathtt{R}}$
where	$\mathtt{S}$ is $\exists \overline{\mathtt{x}} : \overline{\mathtt{R}}. \mathtt{C}\{\mathtt{self} == \mathtt{new\ C}(\overline{\mathtt{x}})\}$
R-CAST	$\mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T}$

- $\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{f}_i \to \mathtt{e}'$ by rule R-FIELD

T-NEW	$\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{V}}$
	$\mathsf{fields}(\mathtt{C}) = \overline{\mathtt{f}} : \overline{\mathtt{F}}$
	$\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{V}} \vdash \overline{\mathtt{x}} :: \overline{\mathtt{F}}$
	$\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{v}}) : \mathtt{R}$
where	$\mathtt{R}$ is $\exists \overline{\mathtt{x}} : \overline{\mathtt{V}}. \mathtt{C}\{\mathtt{self} == \mathtt{new\ C}(\overline{\mathtt{x}})\}$
R-FIELD	$\mathtt{e}'$ is $\mathtt{v}_i$
T-FIELD	$\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{f}_i : \mathtt{T}$
where	$\mathtt{T}$ is $\exists \mathtt{r} : \mathtt{R}. \mathtt{F}_i\{\mathtt{self} == \mathtt{r.f}_i\}$
T-VAR	$\Gamma, \mathtt{y} : \mathtt{V}_i \vdash \mathtt{y} : \mathtt{V}_i\{\mathtt{self} == \mathtt{y}\}$
let t be	$\mathtt{new\ C}(\overline{\mathtt{v}}[\mathtt{y}/\mathtt{v}_i])$
T-NEW	$\Gamma, \mathtt{y} : \mathtt{V}_i \vdash \mathtt{t} : \mathtt{R}'$
where	$\mathtt{R}'$ is $\exists \overline{\mathtt{x}} : \overline{\mathtt{V}}[\mathtt{V}_i\{\mathtt{self} == \mathtt{y}\}/\mathtt{V}_i]$. $\mathtt{C}\{\mathtt{self} == \mathtt{new\ C}(\overline{\mathtt{x}})\}$
	$\sigma(\Gamma) \vdash \mathtt{y} == \mathtt{t.f}_i$ in $\mathcal{X}$
X-PROJ	$\Gamma, \mathtt{y} : \mathtt{V}_i \vdash \mathtt{y} == \mathtt{t.f}_i$
S-CONST-R	$\Gamma, \mathtt{y} : \mathtt{V}_i \vdash \mathtt{y} :: \mathtt{F}_i\{\mathtt{self} == \mathtt{t.f}_i\}$
S-EXISTS-R	$\Gamma, \mathtt{y} : \mathtt{V}_i \vdash \mathtt{y} :: \exists \mathtt{r} : \mathtt{R}'. \mathtt{F}_i\{\mathtt{self} == \mathtt{r.f}_i\}$
Lemma A.8	$\Gamma, \mathtt{y} : \mathtt{V}_i \vdash \mathtt{y} :: \mathtt{T}$

- $\mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{w}}) \to \mathtt{e}'$ by rule R-INVK

R-INVK	$\mathsf{method}(\mathtt{C}, \mathtt{m}) = \mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M} = \mathtt{e}$
	$\mathtt{e}'$ is $\mathtt{e}[\mathtt{new\ C}(\overline{\mathtt{v}})/\mathtt{this}, \overline{\mathtt{w}}/\overline{\mathtt{f}}]$
OK-METHOD	$\mathtt{r} : \mathtt{C}, \overline{\mathtt{f}} : \overline{\mathtt{F}}, \mathtt{c} \vdash \mathtt{e} : \mathtt{E}$
	$\mathtt{r} : \mathtt{C}, \overline{\mathtt{f}} : \overline{\mathtt{F}}, \mathtt{c}, \mathtt{x} : \mathtt{E} \vdash \mathtt{x} :: \mathtt{M}$
T-NEW	$\Gamma \vdash \overline{\mathtt{v}} : \overline{\mathtt{V}}$
	$\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{v}}) : \mathtt{R}$
where	$\mathtt{R}$ is $\exists \overline{\mathtt{x}} : \overline{\mathtt{V}}. \mathtt{C}\{\mathtt{self} == \mathtt{new\ C}(\overline{\mathtt{x}})\}$
Lemma A.5	$\Gamma \vdash \mathtt{C.m} \longrightarrow \mathtt{C}'.\mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M}$
T-INVK	$\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{w}}) : \mathtt{T}$
	$\Gamma \vdash \overline{\mathtt{w}} : \overline{\mathtt{W}}$
	$\Gamma, \mathtt{r} : \mathtt{C}, \overline{\mathtt{f}} : \overline{\mathtt{W}} \vdash \mathtt{c}, \overline{\mathtt{f}} :: \overline{\mathtt{F}}$
where	$\mathtt{T}$ is $\exists \mathtt{r} : \mathtt{R}. \exists \overline{\mathtt{f}} : \overline{\mathtt{W}}. \mathtt{M}$
Lemma A.6	$\Gamma \vdash \mathtt{e}' : \mathtt{S}$
	$\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \exists \mathtt{r} : \mathtt{R}. \exists \overline{\mathtt{f}} : \overline{\mathtt{W}}. \mathtt{E}$
	$\Gamma, \mathtt{r} : \mathtt{R}, \overline{\mathtt{f}} : \overline{\mathtt{W}}, \mathtt{x} : \mathtt{E} \vdash \mathtt{x} :: \mathtt{M}$
Lemma A.9	$\Gamma, \mathtt{x} : \exists \mathtt{r} : \mathtt{R}. \exists \overline{\mathtt{f}} : \overline{\mathtt{W}}. \mathtt{E} \vdash \mathtt{x} :: \mathtt{T}$
S-TRANS	$\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T}$

□

Theorem A.11 (Type Soundness)**.** *If* $\mathtt{P}$ *is well typed* $\vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e}$ *reduces to a normal form* $\mathtt{e}'$ *then either* $\mathtt{e}'$ *contains a stuck cast sub-expression of the form* $\mathtt{v\ as\ T}_0$ *or* $\mathtt{e}'$ *is a value* $\mathtt{v}$ *and there exists* $\mathtt{S}$ *such that* $\vdash \mathtt{v} : \mathtt{S}$*. Moreover, in that case,* $\mathtt{x} : \mathtt{S} \vdash \mathtt{x} :: \mathtt{T}$.

Proof. Straightforward by Theorems A.3 and A.10. □

Theorem A.12 (Method guards)**.** *If* $\mathtt{P}$ *is well typed and* $\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{a}}) : \mathtt{T}$ *and* $\mathtt{e} \to^* \mathtt{new\ C}(\overline{\mathtt{v}})$ *and* $\overline{\mathtt{a}} \to^* \overline{\mathtt{w}}$ *and* $\mathsf{method}(\mathtt{C}, \mathtt{m}) = \mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M} = \mathtt{e}$ *then* $\Gamma \vdash \mathtt{c}[\mathtt{new\ C}(\overline{\mathtt{v}}), \overline{\mathtt{w}}/\mathtt{this}, \overline{\mathtt{f}}]$.

Proof. Using subject reduction and overriding rules.

T-INVK	$\Gamma \vdash \mathtt{e} : \mathtt{E}, \overline{\mathtt{a}} : \overline{\mathtt{A}}$
	$\Gamma \vdash \mathtt{E.m} \longrightarrow \mathtt{C.m}(\overline{\mathtt{f}} : \overline{\mathtt{G}})\{\mathtt{d}\} : \mathtt{N}$
	$\Gamma, \mathtt{x} : \mathtt{E}, \overline{\mathtt{f}} : \overline{\mathtt{A}} \vdash \mathtt{d}, \overline{\mathtt{f}} :: \overline{\mathtt{G}}$
Th. A.10	$\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{v}}) : \mathtt{R}, \overline{\mathtt{w}} : \overline{\mathtt{W}}$
	$\Gamma, \mathtt{x} : \mathtt{R} \vdash \mathtt{x} :: \mathtt{E}$
	$\Gamma, \overline{\mathtt{f}} : \overline{\mathtt{W}} \vdash \overline{\mathtt{f}} :: \overline{\mathtt{A}}$
T-NEW	$\mathtt{R}$ is $\exists \overline{\mathtt{y}} : \overline{\mathtt{W}}. \mathtt{C}\{k\}$
Lemma A.5	$\Gamma \vdash \mathtt{R.m} \longrightarrow \mathtt{C.m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M}$
Lemma A.4	$\Gamma \vdash \mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{F}})\{\mathtt{c}\} : \mathtt{M} \ll \mathtt{m}(\overline{\mathtt{f}} : \overline{\mathtt{G}})\{\mathtt{d}\} : \mathtt{N}$
OK-METHOD	$\Gamma, \mathtt{x} : \mathtt{R}, \overline{\mathtt{f}} : \overline{\mathtt{G}}, \mathtt{d} \vdash \mathtt{c}$
Lemma A.7	$\Gamma, \mathtt{x} : \mathtt{R}, \overline{\mathtt{f}} : \overline{\mathtt{W}} \vdash \mathtt{c}$
	$\Gamma \vdash \mathtt{c}[\mathtt{new\ C}(\overline{\mathtt{v}}), \overline{\mathtt{w}}/\mathtt{this}, \overline{\mathtt{f}}]$

□

Theorem A.13 (Class invariants)**.** *If* $\mathtt{P}$ *is well typed and* $\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{T}$ *and* $\overline{\mathtt{e}} \to^* \overline{\mathtt{v}}$ *then* $\Gamma \vdash \mathsf{inv}(\mathtt{c})[\overline{\mathtt{v}}/\mathtt{this.f}]$.

Proof. Similar to the proof of Theorem A.12. □