Constrained Kinds

Nathaniel Nystrom — University of Texas, Arlington — nystrom@uta.edu

Olivier Tardieu, Igor Peshansky, Vijay Saraswat — IBM T.J. Watson Research Center — {tardieu,igorp,vsaraswa}@us.ibm.com

Abstract

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

In earlier work, we presented the framework of *constrained types* for concurrent, object-oriented languages, parametrized by an underlying constraint system $\mathcal{X}$. Constraint systems are a very expressive framework for partial information. Types are viewed as formulas `C{c}` where `C` is the name of a class or an interface and `c` is a constraint in $\mathcal{X}$ on the immutable instance state of `C` (the *properties*). 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 extend the constraint system 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. Generic types are supported by introducing type variables and permitting programs to impose constraints on such variables.

To illustrate the underlying theory, we develop a formal family of programming languages with a common set of sound type-checking rules parametrized on a constraint system. By varying the constraint system and by extending the typing rules in a simple way, we obtain languages with the power of FJ, FGJ, and languages that provide dependent types, structural subtyping, and constraints that relate values and types. The core of the X10 language is a concrete instantiation of the framework. We describe the design of X10, which is available for download at `x10-lang.org`.

1. Introduction

Modern architectural advances are leading to the development of complex computational systems, such as heterogeneous multi-core, hardware accelerators, and large CPU-count hybrid clusters. The X10 programming language [51, 10, 50] was designed to address the challenge of developing high-performance applications for such machines, building on the productivity gains of modern object-oriented languages.

X10 requires a rich type system to enable code reuse, rule out a large variety of errors at compile-time, and to generate efficient code. A central data structure in X10 is the dense, distributed, multi-dimensional array. Arrays are defined over a set of indices known as *regions*, and support arbitrary base types and accesses through *points* that must lie in the underlying region. For performance it is necessary that array accesses are bounds-checked statically as far as possible. Further, certain regions (such as polyhedral regions) may be represented particularly efficiently. Hence, if a variable is to range only over polyhedral regions, it is important that this information be conveyed statically (through the type system) to the code generator. To support P-way data parallelism it is often necessary to logically partition an array into P pieces. A type system that can establish that a given division is a partition can ensure that no race conditions arise due to simultaneous accesses by multiple activities to different pieces.

1.1 Constrained types

These requirements motivated us to develop a framework for dependent types in object-oriented languages [44]. *Dependent type systems* [35, 59, 12] have been extensively developed over the past few decades in the context of logic and functional programming—they permit types to be parametrized by *values*.

The key idea behind our approach is to focus on the notion of a *constraint system* [49]. Constraint systems were originally developed to provide a simple framework for a large variety of inference systems used in programming languages, in particular as a foundation for constraint programming languages. Patterned after Scott's information systems, a constraint system is organized around the notion of *constraints* or tokens of partial information (e.g., `x+y>z*3`), together with an entailment relation $\vdash$. Tokens may have first-order structure; existential quantification is supported. The entailment relation is required to support a certain set of inference rules arising from a Gentzen-style formulation of intuitionistic logic.

In applying constraint systems to object-oriented languages[1], the principal insight is that objects typically have some immutable state, and constraints on this state are of interest to the application. For instance, in Java the length of an array might not be statically known, but is fixed once the array is created. Hence, we can enrich the notion of a type: for a class `C` we permit a *constrained type* `C{c}` where `c` is a constraint on the immutable fields, or *properties*, of the class as well as any immutable variables and constants in scope [44]. Constraints are drawn from a constraint language that, syntactically, is a subset of the boolean expressions of X10. For brevity, the constraint may be omitted and interpreted as `true`. Thus, `Point{self.rank==N}` is a type satisfied by any `N`-dimensional point, that is, any instance of `Point` whose `rank` property is `N`. `N`, here, 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 `Array`. Subtyping is easily defined: a type `C{c}` is a subtype of `D{d}` provided that `C` is a subclass of `D` and `c` entails `d` in the underlying constraint system.

Constrained types maintain a phase distinction between compile time (entailment checking in the underlying constraint system) and run time (computation). Dynamic type casting is permitted—code is generated to check at run time that the properties of the given object satisfy the given constraint. We note that while constraints may reference only immutable state, a constrained type can classify mutable variables. For instance the mutable field `tail` of the class `List` may be typed with `List{self.length==N}` and will be

[1] The use of constraints for types has a distinguished history going back to Mitchell [39]. Our work is closely related to the HM(X) approach [52]—see Section 2.6 for details.

statically checked to ensure that at runtime it can only contain lists that satisfy the given property.

The constrained types approach enjoys many nice properties in contrast to similar approaches such as DML [59]. Constrained types are a natural extension to OO languages, and quite easy to use. Constraints may also be used to specify class invariants, and conditions on the accessibility of fields and methods (conditional fields and methods). Final variables in the computation can be used directly in types; there is no need to define a separate parallel language of index expressions to be used in the type system. Constrained types always permit field selection and equality at object types; hence the programmer may specify constraints at any user-specified object type, not just over the built-in constraint system.

1.2 Generic types

The notion of *generic types*—types such as `List<T>` in Java that are parametrized by other types—is now widely established [29, 42, 5, 40, 54, 21, 53]. Generic types are vital for implementing type-safe, reusable libraries, especially collections classes. For instance, the data type `Array` discussed above is generic on its element type.

This paper lays out a framework to extend constrained types to handle (type-)genericity. The general outlines of the approach seem clear enough. First, some mechanism must be found to introduce *type variables*, for instance as *type parameters* (cf. Java [21]) or as *type members* (cf. BETA [31]).

Second, a suitable vocabulary of constraints over types must be chosen. This raises the fundamental question: What is the structure of types in (nominal) OO languages? The answer to this is fairly clear. A type is a *name* (e.g., class name or interface name), there is a partial order (the *inheritance* relation) on such names, and an association of (typed) members (fields, methods) to these names. This structure raises some natural candidates for constraints: e.g., `X <= T` (realized by any type `S` that inherits from `T`), `X` $\underline{\text{has}}$ `m(`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`):T`, (realized by any type `S` that has a method named `m` with the given signature), `x instanceof X` (realized by any type `S` to which the final variable `x` can be cast), etc.

Third, these constraints can be used to specify *constrained kinds* in the same way that constraints over values can be used to specify *constrained types*. Thus `X:*{self <= Printable}` would declare a type variable `X` which could only be bound to those types `S` which satisfy `S <= Printable`.

Fourth, a type variable could then be declared as a (kinded) class property, field, or method parameter. Within the scope of its declaration, the type variable can be used wherever a type can (e.g., to specify the type of method parameters and return values, local variables, fields, as targets of cast etc).

While the outline is clear, a number of important issues remain untouched:

- What is the difference in expressiveness between type parameters and type properties?
- How do constraints on types interact with constraints on values?
- Should constrained kinds be used to differentiate overloaded methods? During dynamic method lookup?
- Can constrained kinds be used to express variance declarations (e.g., contravariance, covariance, invariance)? Java "wildcards"?
- How are constrained kinds implemented? What kind of runtime representation of types is needed, and how does it interact with the constraint solver?

1.3 Contributions

This paper develops the framework of constrained kinds. In the next section we address answers to the questions above, and illustrate with programming examples. We then motivate the choices made in the design of the X10 language, and outline the implementation strategy. An implementation of X10 (realizing these features) can be downloaded from `x10-lang.org`.

We present a formalization of these ideas through an extension of the development in [44]. We present a core calculus, FXG, parametrized on an underlying data constraint system. The calculus can express `X <= T` constraints on type variables. We present subject reduction and type soundness properties for the calculus. This calculus embeds the calculus presented in [44].

We show how the framework can be extended in a simple fashion to handle structural typing constraints.

We summarize our contributions:

- We extend the notion of constrained types [44] with *constrained kinds*. These permit type variables to be classified based on constraints over types and values.
- We explore the design space for an OO language with constrained kinds and types.
- We formalize a core "featherweight" calculus for constrained kinds, permitting subtype constraints, conservatively extending [44] and establish subject reduction and type soundness.
- We show how the framework can be extended in a simple fashion by adding other constraints on types (and associated typing rules).

Outline. The rest of the paper is organized as follows. Design considerations for generics in X10 and related work are discussed in Section 2. Section 3 presents a formal semantics and a proof of soundness. Finally, Section 4 concludes.

2. Design considerations

X10 is a class-based object-oriented language that provides both dependent and generic types. The language has a sequential core similar to Java or Scala, but also constructs for concurrency and distribution.

A key feature that interacts with generics is that the type system provides both reference types and value types. An instance of a reference type is an object on the heap. All reference types are subclasses of `Object`. Variables of reference type may be `null`. In contrast, an instance of a value type might be represented in unboxed form on the call stack and can never be `null`. Ideally, the design should support instantiation of generics on both reference and value types.

This section describes the design of generics for X10, including several alternative designs. These alternatives demonstrate the expressiveness of constrained kinds.

2.1 Type constraints

To permit genericity, variables `X` must be admitted over types. The choice of type variables is discussed below. We assume here that classes have a means of introducing new type variables either as type parameters or as type members. For instance, the class `List` introduces a type variable `X` representing the list's element type.

X10 already supports constraints over values, so it is natural to extend these to constraints over types. Here, we ask: how should type variables be constrained?

Constraints occur in several places in the X10 syntax. They are of course permitted in constrained types `C{c}`. Constraints may also be used as *class invariants*, which are constraints on the class's properties and other final variables in scope. The class invariant must be established by the class's constructor and subsequently holds for all instances of the class.

Methods and constructors may also have constraints, or *guards*, on their parameters. A guard must be satisfied by the caller of the method and will hold throughout its body. Type constraints used in the method guard restrict the types of the arguments or of the method receiver.

2.1.1 Nominal subtyping and equality constraints

In class-based OO languages such as Java, types are equipped with a partial order (the *subtyping* order) generated from the user program through the "`extends`" relationship. This motivates a very natural constraint system on types. For a type variable `X` we should be able to assert the constraint `X <= T`: a valuation (mapping from variables to types) realizes this constraint if it maps `X` to a type that extends `T`. Constraints on types can specify either subtype (<=), supertype (>=), or equality bounds (==).

Using subtyping constraints in the class invariant provides a means to bound the type variables introduced by the class declaration. Constraints in constrained types `C{c}`, can bound type-valued members of the base type `C`.

These constraints also can be used in method guards. This feature is similar to optional methods in CLU [29] and to generalized type constraints in C♯ [14]. 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.

2.1.2 Structural constraints

One should also be able to 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(`$\overline{\texttt{x}}$`:`$\overline{\texttt{S}}$`):T` to express this. These constraints allow one to define an alternative version of the `print` methods above as:

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

Rather than restricting the actual receiver to lists whose element type implements `Printable`, with structural constraints, any list whose element type has a `print` method may be used.

Structural constraints on types are found in many languages. For instance, Haskell supports type classes [26, 23]. 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 [34] is a Java-like language with both nominal and structural subtyping.

In the class invariant, a structural constraint can bound the class's type variables, similarly to the language PolyJ [40], which allows type parameters to be bounded using structural *where clauses* [13]. For example, a sorted list class could be written as follows in PolyJ:

```
class SortedList[T] where T {int compareTo(T)} {
  void add(T x) {... x.compareTo(y) ...}
  ...
}
```

The `where` clause states that the type parameter `T` must have a method `compareTo` with the given signature. `SortedList` can be instantiated on any type that provides the method. With nominal bounds, `SortedList` could only require that its parameters implement an interface such as `Comparable`.

2.1.3 Default values

Recall that X10's type system provides both reference types and value types. In languages like Java with primitive types, every type has a default value—`null` for reference types, `false` or `0` for primitive types—used to initialize arrays of that type. In X10, some types do not have an obvious default value. For example, `int{self>0}` does not contain the value `0`. Consequently, a useful constraint is `T has default`, which holds if the type `T` has a default value.

2.1.4 `instanceof` constraints

Lastly, we consider constraints of the form `x instanceof T`. By relating types and values in a single constraint, these constraints provide considerable expressive power. For instance, consider the class declaration:

```
class C {
  def equals[T](x:T) {this instanceof T} = (this==x);
}
```

The `equals` method can be called with any object that is a supertype of `C`.

`instanceof` constraints can be used to build intersection types, e.g., `Object{self instanceof A,self instanceof B}` should be a subtype of `A` and `B`..

2.2 Type variables

Languages such as Java [21] and Scala [45] introduce *type parameters* on classes and methods. An alternative approach, used by BETA [31], Scala [45], and other languages is, to use type members. In X10, one can generalize properties to include type-valued properties: A *type property* is a final object member initialized at construction time with a concrete type.

2.2.1 Type properties

Like normal value properties, type properties can be used in constrained types through the variable `self`. This immediately suggests use-site variance constraints [55, 25] on type properties. The type of a list of integers, say, can be written as `List{self.T==int}`. Nominal subtyping constraints, then, may be used to provide use-site variance constraints. Consider the following subtypes of `List` with type property `T`:

- `List`. This type has no constraints on the type property `T`. Any type that constrains `T` is a subtype of `List`.
- `List{T==float}`. The type property `T` is bound to `float`. For a final expression `v` of this type, `v.T` and `float` are equivalent types and can be used interchangeably.
- `List{T<=Collection}`. This type constrains `T` to be a subtype of `Collection`. All instances of this type must bind `T` to a subtype of `Collection`; for example `List[Set]` (i.e., `List{T==Set}`) is a subtype of `List{T<=Collection}` because `T==Set` entails `T<=Collection`.
- `List{T>=String}`. This type bounds the type property `T` from below.

While expressive, type properties have a number of usability issues. The key difference between type parameters and type properties is that type properties are instance *members* bound during object construction. Type properties are thus accessible through expressions—`e.T` is a legal type (if `e` is final)—and are inherited by subclasses. These features give type properties more expressive power than type parameters, as we shall describe below; however, because they provide similar functionality with often subtle distinctions, type properties can be difficult to use, especially for novices, and require more care in the design. For instance, since type properties are inherited, the language design needs to account for ambiguities introduced when the same name is used for different type properties declared in or inherited into a class.

Virtual types. Type properties provide expressive power much like *virtual types* [31, 32, 15]; moreover, they can also be constrained at the use-site, can be refined on a per-object basis without explicit subclassing, and can be refined contravariantly as well as covariantly.

Thorup [54] proposed adding genericity to Java using virtual types. For example, a generic `List` class can be written as follows:

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

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

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

Only classes where `T` is final bound, such as `IntList`, can be non-abstract. Scala [45] supports abstract types and virtual types in a similar way. The analogous definition of `List` in X10 using type properties is as follows:

```
class List(T:*) {
  def get(i: int): T { ... }
}
```

Unlike the virtual-type version, the X10 version of `List` is not abstract; `T` need not be instantiated by a subclass because it can be instantiated (constrained) on a per-object basis. Rather than declaring subclasses of `List`, one uses the constrained subtypes `List{T<=Number}` and `List{T==Integer}`.

Type properties can also be refined contravariantly. For instance, one can write the type `List{T>=Integer}`.

Self types. Type properties can also be used to support a form of self type [6, 7]. Self types can be implemented by introducing a type property `type` to the root of the class hierarchy, `Object`:

```
class Object(type:*){type <= Object} { ... }
```

For any final path expression `p`, the type `p.type` represents all instances of the fixed, but statically unknown, run-time class referred to by `p`. Scala's path-dependent types [45] and J&'s dependent classes [43] take a similar approach.

Self types are achieved by constraining types so that if a path expression `p` has type `C`, then `p.type<:C`. In particular, one can add the class invariant `this.type<:C` to every class `C`. This invariant ensures that `this.type` is a subtype of the lexically enclosing class.

The property must be initialized to the given class, so, without further language support, one must create an instance of `Object` with `new Object(Object)` to initialize the `type` property.

2.2.2 Type parameters

Most OO languages provide genericity through type parameters on classes and methods. The development of a nominal OO type system with type parameters is now standard (cf. FGJ [24]).

Scala [45] supports definition-site variance annotations: a parameter may be declared in-, co-, or contravariant. If the parameter `X` of a class `C` is covariant, then `S` a subtype of `T` implies `C[S]` is a subtype of `C[T]`. Similarly, if `X` is contravariant, `C[T]` is a subtype of `C[S]`. Invariant parameters are the default; a covariant parameter is declared by prepending "+" to the parameter name in the class header; a contravariant parameter is declared by prepending "–". The usage of variant parameter types in the body of their class must be restricted to ensure the subtyping relation holds.

Java, by contrast, supports use-site variance through wildcards. This has a number of usability problems [30], which also occur with constrained type properties, above.

2.3 Overloading and dispatch

The next question to address is the overloading semantics for methods with constraints on formal parameters and with method guards. This issue was considered in non-generic X10 but was revisited in light of type constraints.

One option is to ignore constraints when checking for overloading. This means that `m(int{self==0})` and `m(int{self==1})`, for instance, are considered to have the same signature; if both occur within the same class, a compile-time error occurs.

Another option is to allow the overloading: methods are resolved at compile-time, based on the constraints. 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.

Allowing the overloading on constraints can also complicate method overriding by introducing partial overrides. Consider:

```
class A {
  def m(x:int{self<=0}) = ...; // 1
  def m(x:int{self>=0}) = ...; // 2
}
class B {
  def m(x:int{self==0}) = ...; // 3
}
```

`B.m`'s constraint on `x` partially overrides the constraint on both `m` methods of `A`. Given a variable `b` of type `B`: `b.m(-1)` invokes `A.m` (method 1), `b.m(0)` invokes `B.m` (method 3), and `b.m(1)` invokes `A.m` (method 2). Clients of `B` could get confused about which method gets invoked. One option is to require that when a method with a given name is overridden, all other methods with that name should be overridden as well.

Finally, one could support a form of predicate dispatch [38], selecting the method to invoke by *dynamically* evaluating the method guard. With type constraints and predicate dispatch, multi-method dispatch can be implemented.

2.4 Implementation

Finally, we turn to the implementation of generics. To implement a generic class `C[X]` one can either generate a single class for `C` in the target language (homogeneous translation) or generate one class per instantiation $\texttt{C[T}_1\texttt{]}, \ldots, \texttt{C[T}_k\texttt{]}$ (heterogeneous translation). The former approach reduces the amount of generated code; the latter enables specialization based on the type arguments to `C`. Hybrid approaches are possible as well.

Java's approach is to erase type parameters and to use the homogeneous translation. 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. The homogeneous translation is aided by a restriction that type parameters cannot be instantiated on primitive types and by using nominal subtyping bounds on types. These restrictions ensure type parameters can be represented with their type bound, or `Object` if unbounded. Since, in X10, it should be possible to instantiate a generic type on both value types and reference types, a homogeneous translation must box value types so that both kinds of types have the same representation.

PolyJ [40] supports structural bounds and uses a homogeneous translation with adapter objects to allow generic code to invoke methods on values of its type parameters.

Other languages, such as C++, use a heterogeneous translation, specializing the generic class for each instantiation. C♯, NextGen [9], and Fortress [2] takes this approach as well, reducing (static) code bloat by instantiating generic classes at run time.

A compromise approach is to specialize for only a few parameter types, for example the primitive types, but to use a homogeneous translation otherwise.

Representing type variables at run-time allows the language to support run-time casts to generic types, including possibly types instantiated on constrained types.

With non-generic types, a cast such as `r as Region{rank==k}` can be implemented by checking the run-time class of the value being cast—`r instanceof Region`—and then evaluating the constraint—`r.rank==k`. However, the issue is more subtle with generic casts. For instance, to do `A as Array[int{self>=0}]` one must check at run time that the concrete type used to instantiate the `Array`'s type parameter is equivalent to `int{self>=0}`. This check could involve a run-time entailment check, breaking the phase distinction between compile time and run time for constraint solving.

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. Finally, one can simply erase the constraints from the run-time type information, preserving the base type. As with Java's erasure semantics, this approach is prone to run-time type errors.

2.5 X10 design decisions

Given these considerations, the X10 makes the following choices:

- X10 supports subtyping and equality constraints on types
- X10 does not support structural bounds, but may do so in the future. X10 has closures with structural subtyping, which can be used in many of the cases structural type bounds would be used.
- Classes have type parameters with definition-site variance rather than type properties with use-site variance annotations. Properties are just too unfamiliar. Usability outweighs expressive power.
- Run-time type information is preserved, but constraints are not.

2.6 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 [44].

Constraint-based type systems. The use of type constraints for type inference and subtyping was first proposed by Mitchell [39] and Reynolds [47]. HM(X) [52] 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 [59, 36, 3] parametrize types on values. Our work is closely related to Dependent ML (DML [59]), 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 [19]: 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 are introduced; in X10, constraints are defined over program variables and types.

Logically qualified types, or liquid types [48], 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.

Another dependent-type approach is to integrate theorem provers into the programming language. Concoqtion [18] extends types in OCaml [28] with constraints written as Coq [11] rules. While the types are expressive, supporting the full generality of the Coq language, proofs must be provided to satisfy the type checker. X10 supports only constraints that can be checked by a constraint solver during compilation. Ynot [41] is an extension to Coq for reasoning about dependently-typed functional programs with side-effects using a form of separation logic. Constrained types permit a more natural expression of constraints than Coq formulas; however, unlike Ynot, they do not capture side-effects since they constrain only immutable object state.

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 [5, 40, 54, 1]. GJ [5] implements invariant type parameters via type erasure. Java5 [21] adopted the same implementation approach, incorporating wildcards and raw types. Other proposals [40, 9, 58, 57] support run-time representation of type parameters. PolyJ [40] permits instantiation of parameters on primitive types and structural parameter bounds. MixGen [1] supported mixins through type parametrization.

Like X10's generics, Scala [45]'s generic types provide definition-site variance annotations and polymorphic methods. Type parameters definitions can specify bounds. Scala also provides mixins and self types, abstract type members, and virtual types.

Fortress [2] also support generic types. Traits and objects may be parametrized on values as well as types. Parametrization on values is limited to integers, naturals, booleans, and operators that are compile-time constants. Value properties in X10 can be of arbitrary type and can be initialized with arbitrary values; run-time constants (i.e., final expressions) can be used in constraints. Fortress allows parameters to be hidden, reducing the programmer burden by allowing the compiler to infer the arguments.

Virtual classes and *virtual types* [31, 32, 15] are another mechanism for supporting genericity, using nested types rather than parametrization. Thorup [54] proposed using virtual types to provide genericity in Java, and much of the developemnt of Java's generics followed from virtual classes: use-site variance based on structural virtual types was proposed by Thorup and Torgersen [55] and extended for parametrized type systems by Igarashi and Viroli [25]; the latter type system lead to the development of wildcards in Java [21, 56, 8]. Dependent classes [20] generalize virtual classes to express similar semantics via parametrization rather than nesting. Virtual classes depend only on their enclosing instance; dependent classes, in contrast, depend on all the objects in which they are parametrized. 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.

C$^\sharp$ also supports generic types via run-time instantiation in the CLR [53]. Type parameters may be declared with definition-site variance tags. Generalized type constraints were proposed for C$^\sharp$ [14]: methods can be annotated with subtyping constraints that must be satisfied to invoke the method. Generic X10 supports these constraints, as well as constraints on values, with method and constructor where clauses.

Contracts. Constrained types have many similarities to contracts [46, 37, 16], constraint annotations on methods and classes. A key difference is that contracts are dynamically enforced assertions; whereas, constrained types are statically enforced. Dynamic enforcement enables a more expressive assertion language at the cost of possible run-time failures. Integration of constrained types and generic types effectively allow parametrization of classes on constraints. Consider a `List` class parametrized on the element type. If the class is instantiated on a constrained type `C{c}`, i.e., `List[C{c}]`, then all elements of the list satisfy `c`. Guha et al. [22] explore polymorphic contracts as a way to achieve this parametrization in a functional language. Generic constrained types provide constraint parametrization in a natural extension of object-oriented languages.

ESC/Java [17], Spec# [4], and JML [27] are instances of static assertion checkers for OO languages. Constrained types offer a less expressive, but more modular, lightweight solution.

3. Semantics

In the previous section we presented X10. The subtle issues encountered when designing and implementing the X10 type system exposed the need for a formal framework in which to explore the design space, and to reason about fundamental issues such as soundness, completeness, and decidability. The resulting framework consists of a family of formal languages—the FXG family—that share a common base language, operational semantics, and type system.

Our framework models many, but not all, relevant features of X10. Following FJ, our framework does not account for mutations. Proving the soundness of the type system for an imperative calculus is beyond the scope of this paper. Other features of X10 not modeled in the framework include interfaces, constructors, and method overloading. Class invariants are simplified and may constrain only type parameters, but not properties of the class. We believe none of these restrictions affect the type system in a fundamental way. Finally, our subtyping relation is invariant in the type parameters. We leave modeling the definition-site variance annotations of X10 for future work.

We first describe the base FXG language and its type system and prove it sound. While its FGJ-like syntax enables the declaration of generic classes and methods, it does not allow constraints on type parameters. We then demonstrate other languages in the family with features such as FGJ-like generic types, or structural subtyping constraints.

3.1 The FXG language

The grammar for FXG is shown in Figure 1. The syntax is essentially that of FGJ. We use $\bar{x}$ to denote a list $x_1, \ldots, x_n$, and use $\bullet$ to denote the empty list.

A program $\mathtt{P}$ is a set of class declarations $\overline{\mathtt{L}}$. Class names $\mathtt{C}$ range over the declared classes in `P` and `Object`. A class declaration has type parameters $\overline{\mathtt{X}}$, value properties (i.e., immutable fields) $\overline{\mathtt{f}}$, a supertype $\mathtt{D}[\overline{\mathtt{E}}]$, methods $\overline{\mathtt{M}}$, and a guard `c`—a constraint on its type parameters.

Each class has a default constructor. If class $\mathtt{C}[\overline{\mathtt{X}}]$ has field `f` of type `F` and extends class `D` with field `g` of type `G` then `C`'s constructor has type parameters $\overline{\mathtt{X}}$, a formal of type `G`, and a formal of type `F`. `Object` has a nullary constructor.

Methods are introduced with the `def` keyword. A method has both type parameters $\overline{\mathtt{X}}$ and value parameters $\overline{\mathtt{x}}$. The method guard `c` is to be thought of as an additional condition that must be satisfied by the receiver and the actual type and value arguments of the method call. We do not consider method overloading: we assume each class declares at most one method with name `m`.

Square brackets may be dropped in the absence of type parameters, as well as `true` constraints and the surrounding curly braces.

The body of a method is an expression `e`. It is built from the value parameters of the method (including the receiver `this`), field access expressions, constructor calls, method invocations, and casts (written `e as G`).

The set of types includes class types $\mathtt{C}[\overline{\mathtt{A}}]$, type parameters `X`, dependent types ($\mathtt{T}\{\mathtt{c}\}$), and is closed under existential quantification ($\exists \mathtt{x}{:}\mathtt{T}.\ \mathtt{U}$). Existential types arise in typing judgments but are not permitted in programs. Neither casts nor methods invocations are permitted in constraints. A value `v` is of type $\mathtt{C}[\overline{\mathtt{A}}]$ if it is an instance of class $\mathtt{C}[\overline{\mathtt{A}}]$; it is of type $\mathtt{T}\{\mathtt{c}\}$ if it is of type `T` and it satisfies the constraint $\mathtt{c}[\mathtt{v}/\mathtt{self}]$; it is of type $\exists \mathtt{x}{:}\mathtt{T}.\ \mathtt{U}$ if there exists some value `w` of type `T` such that `v` is of type $\mathtt{U}[\mathtt{w}/\mathtt{x}]$. In other words, the set of values of $\exists \mathtt{x}{:}\mathtt{T}.\ \mathtt{U}$ is the union of the sets of values $\mathtt{U}[\mathtt{w}/\mathtt{x}]$ for all `x` of type `T`.

Following FJ, we denote values by means of nested constructor calls, e.g., `new Pair[C](new C(), new C())`.

Dynamic semantics. The operational semantics, shown in Figures 2 and 3, is described as a reduction relation on expressions. It enforces a strict left-to-right call-by-value evaluation order.

The use of the subtyping relation to check that the cast is satisfied is the only reason the semantics has to keep track of type parameters. The existence of fields and methods of value $\mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{v}})$ do not depend on the types $\overline{\mathtt{A}}$. Moreover, even if the applicability of a method depend on the guard hence the type parameters, this is irrelevant to the operational semantics because we will establish that there is no need for a run-time check of the guards in well-typed programs. As a consequence, R-INVK does not check method guards.

L-FIELD-I prevents classes from overriding inherited fields. L-METHOD-I makes sure method lookup goes bottom up, but does not enforce any overriding restrictions. These are not relevant to the operational semantics, hence are introduced later.

R-CAST checks that the static type of $\mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{v}})$ is a subtype of `G`.

Conventions. In the following, the context Γ is always a (finite, possibly empty) sequence of variable declarations $\mathtt{x}{:}\mathtt{T}$, type parameter declarations $\mathtt{X}{:}*$, and constraints `c` satisfying:

1. for any declaration $\phi = \mathtt{x}{:}\mathtt{T}$ or constraint $\phi = \mathtt{c}$ in the sequence, all free variables[2] occurring in `T` or `c` are declared in the sequence to the left of ϕ.

[2] Variables are bound by existential quantifiers. In addition, the type constructor $\mathtt{c} \mapsto \mathtt{T}\{\mathtt{c}\}$ binds the special variable `self` in `c`.

2. a variable or type parameter is declared at most once in Γ.

For any formulas ϕ_1 and ϕ_2, the judgment $\Gamma \vdash \phi_1,\ \phi_2$ is shorthand for the judgments $\Gamma \vdash \phi_1$ and $\Gamma \vdash \phi_2$.

An assumption "x fresh" in an inference rule means that x does not occur in the premises to the left of this assumption.

A premise $\theta = \overline{\alpha}/\overline{\beta}$ requires $\overline{\alpha}$ and $\overline{\beta}$ to have the same length n and defines the substitution of β_i by α_i for $1 \leq i \leq n$. If ϕ is a constraint, a type, an expression, etc., $\phi\theta$ denotes the result of the application of the substitution θ to the free variables of ϕ.

Type system. Typing checking a program P involves a series of judgments:

1. Constraints:
   $\Gamma \vdash \mathtt{c}$ — Γ entails constraint c
2. Well-formedness:
   $\Gamma \vdash_{\mathcal{T}} \mathtt{t}$ — constraint term t is well-formed in Γ
   $\Gamma \vdash_{\mathcal{C}} \mathtt{c}$ — constraint c is well-formed in Γ
   $\Gamma \vdash_{\mathcal{W}} \mathtt{T}$ — type T is well-formed in Γ
3. Lookup:
   $\Gamma \vdash \mathtt{x\ has\ I}$ — variable x has member I in Γ where $\mathtt{I} ::= \mathtt{f:F} \mid \mathtt{m[\overline{B}](\overline{x}:\overline{G})\{c\}:H}$
4. Subtyping:
   $\Gamma, \mathtt{x:S} \vdash \mathtt{x <: T}$
   type $\mathtt{S\{self == x\}}$ is a subtype of type T in $\Gamma, \mathtt{x:S}$
5. Typing:
   $\Gamma \vdash \mathtt{e:T}$ — expression e has type T in Γ
   $\Gamma \vdash \mathtt{I} \ll \mathtt{J}$ — member I overrides member J in Γ
   $\vdash \mathtt{def\ m[\overline{Y}](\overline{x}:\overline{G})\{c\}:H = e}$ OK in $\mathtt{C[\overline{X}]}$
   method m in class $\mathtt{C[\overline{X}]}$ is well-typed
   $\vdash \mathtt{class\ C[\overline{X}]\{c\}(\overline{f}:\overline{F})\ extends\ D[\overline{E}]\ \{\ \overline{M}\ \}}$ OK
   class $\mathtt{C[\overline{X}]}$ is well-typed

Notice these judgments depend on the particular program P. For readability, we do not make this dependency explicit as we will never consider more than one program at a time.

A program is well-typed iff all its classes are. We now describe in more detail each of these judgments, in turn.

1. Constraints. FXG is parametrized by an *object constraint system* $\mathcal{X}$. Such a constraint system is required to have terms t of the form $\mathtt{C(\overline{f} = \overline{t})}$ and t.f, and an equality predicate on such terms. (It may have other predicates and terms whose interpretation is left unconstrained, see Section 3.2.) The entailment relation for $\mathcal{X}$ must respect the interpretation of (a) $\mathtt{C(\overline{f} = \overline{t})}$ as a finite tree with root labeled with C, *i*th 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.[3]

In order to expose the current typing context to $\mathcal{X}$, we define the *constraint projection* $\sigma(\Gamma)$ that, in essence, strips out all type information from the Γ. It also uses information in the program P to translate constraints in the source program into constraints understood by the constraint system.

$\sigma(\varepsilon) = \mathtt{true}$
$\sigma(\mathtt{new\ C[\overline{A}](\overline{t})}) = \mathtt{C(\overline{f} = \overline{t})}$ where $\mathsf{fields}(\mathtt{C[\overline{A}]}) = \mathtt{\overline{f}:\overline{G}}$
$\sigma(\mathtt{t.f}) = \mathtt{t.f}$
$\sigma(\mathtt{f(\overline{t})}) = \mathtt{f}(\sigma(\overline{\mathtt{t}}))$ for all other functions f
$\sigma(\mathtt{t == s}) = \sigma(\mathtt{t}) == \sigma(\mathtt{s})$
$\sigma(\mathtt{p(\overline{t})}) = \mathtt{p}(\sigma(\overline{\mathtt{t}}))$ for all other predicates p
$\sigma(\mathtt{c}, \Gamma) = \sigma(\mathtt{c}), \sigma(\Gamma)$
$\sigma(\mathtt{X}:*, \Gamma) = \sigma(\Gamma)$
$\sigma(\mathtt{x:C[\overline{A}]}, \Gamma) = \sigma(\Gamma)$
$\sigma(\mathtt{x:X}, \Gamma) = \sigma(\Gamma)$
$\sigma(\mathtt{x:T\{c\}}, \Gamma) = \mathtt{c}\theta, \sigma(\mathtt{x:T}, \Gamma)$ where $\theta = \mathtt{x/self}$
$\sigma(\mathtt{x}:\exists\mathtt{y:T.\ U}, \Gamma) = \sigma(\mathtt{z:T}, \mathtt{x:U}\theta, \Gamma)$ where $\theta = \mathtt{z/y}$

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.

We specify that $\Gamma \vdash \mathtt{c}$ if the constraint projection of Γ entails that of c in the input constraint system.

We say that a context Γ is *consistent* if it is not the case that $\Gamma \vdash \mathtt{false}$. In all inference rules presented below, we make the implicit assumption that the context Γ of every premise is consistent; if one is inconsistent, the rule cannot be used. In the sequel, we will permit type system extensions to mark contexts as inconsistent, e.g., X extends class C, X extends class D entails false if C and D are not related by the subclassing relation.

2. Well-formedness. A constraint term, constraint, or type α is well-formed in context Γ iff its free variables are declared in Γ and all the type parameters of all the generic classes in α satisfy the guards of these classes. Note that this depends on $\mathcal{X}$ but not on any typing judgments. The rules of well-formedness of terms and constraints are straightforward and are omitted. The rules for types are specified in Figure 4.

We say a context Γ is well-formed if each α in Γ is well-formed w.r.t. the sequence of Γ to its left, that a judgment is well-formed iff its context is well-formed and the consequent is well-formed w.r.t. the context.

In all inference rules presented below (except for OK-METHOD and OK-CLASS), we make the implicit assumption that every lookup, subtyping, or typing judgment is well-formed. If it is not then the rule cannot be used.

By design, if the program P is well-typed then all the constraints and types in P are well-formed in their respective contexts.

3. Lookup. Figure 5 specifies the field and method signatures available on each type. In $\mathsf{FXG}(\mathcal{X})$, these are exactly those captured by the fields and methods predicates for class types and none for type parameters. Constraints on type parameters will change that.

4. Subtyping. The subtyping relation is defined in Figure 6. Because we deal with dependent and existential types, we choose a somewhat unconventional notation for the subtyping relation that makes formal developments less verbose. Rather than judgments of the form $\Gamma \vdash \mathtt{S <: T}$, we consider judgments of the form $\Gamma, \mathtt{x:S} \vdash \mathtt{x <: T}$ which permit T to depend on x, hence make code-dependent types easier to work with. In fact, one could define classical subtyping as the following:

$$\frac{\Gamma \vdash_{\mathcal{W}} \mathtt{S} \qquad \Gamma, \mathtt{x:S} \vdash \mathtt{x <: T} \qquad \mathtt{x} \text{ not free in } \mathtt{T}}{\Gamma \vdash \mathtt{S <: T}}$$

or derive our relation from subtyping:

$$\frac{\Gamma, \mathtt{x:S} \vdash \mathtt{S\{self == x\} <: T}}{\Gamma, \mathtt{x:S} \vdash \mathtt{x <: T}}$$

The intent of the subtyping relation is to combine subclassing and constraint entailment: type $\mathtt{C[\overline{A}]\{c\}}$ is a subtype of $\mathtt{D[\overline{A}]\{c\}}$ iff C is a subclass of D and c entails d in the underlying constraint system. In the formal model, subtyping is invariant in the type parameters, that is, $\mathtt{C[\overline{A}]\{c\}}$ is a subtype of $\mathtt{C[\overline{B}]\{c\}}$ only if $\overline{\mathtt{A}} = \overline{\mathtt{B}}$.

Rules S-CONST-L and S-CONST-R let us rearrange constraints in types, e.g., $\mathtt{x:T\{c,d\}} \vdash \mathtt{x <: T\{c\}\{d\}}$.

5. Typing. The typing rules are specified in Figure 7.

T-VAR is as expected, except that it asserts the constraint self==x, which records that any value of this type is known statically to be equal to x.

We do away with the three cast rules in FJ in favor of a single cast rule, requiring only that e be of some type T. At run time, e

[3] A complete axiomatization of the algebra of finite trees is provided in [33].

$$
\begin{array}{rrcl}
\text{(Program)} & \mathtt{P} & ::= & \overline{\mathtt{L}} \\
\text{(Class declaration)} & \mathtt{L} & ::= & \mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{G}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{G}}]\ \{\ \overline{\mathtt{M}}\ \} \\
\text{(Method declaration)} & \mathtt{M} & ::= & \mathtt{def}\ \mathtt{m}[\overline{\mathtt{X}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{G} = \mathtt{e}; \\
\text{(Expression)} & \mathtt{a,b,e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{new}\ \mathtt{C}[\overline{\mathtt{G}}](\overline{\mathtt{e}}) \mid \mathtt{e.m}[\overline{\mathtt{G}}](\overline{\mathtt{e}}) \mid \mathtt{e}\ \mathtt{as}\ \mathtt{G} \\
\text{(Constraint term)} & \mathtt{t,u} & ::= & \mathtt{x} \mid \mathtt{t.f} \mid \mathtt{new}\ \mathtt{C}[\overline{\mathtt{G}}](\overline{\mathtt{t}}) \\
\text{(Constraint)} & \mathtt{c,d} & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{c,c} \mid \mathtt{t} == \mathtt{t} \\
\text{(Generic type)} & \mathtt{A,B,E,F,G,H} & ::= & \mathtt{C}[\overline{\mathtt{G}}] \mid \mathtt{G}\{\mathtt{c}\} \mid \mathtt{X} \\
\text{(Type)} & \mathtt{S,T,U} & ::= & \mathtt{C}[\overline{\mathtt{G}}] \mid \mathtt{T}\{\mathtt{c}\} \mid \mathtt{X} \mid \mathtt{T}\{\mathtt{c}\} \mid \exists \mathtt{x}:\mathtt{T}.\ \mathtt{T} \\
\text{(Value)} & \mathtt{v,w} & ::= & \mathtt{new}\ \mathtt{C}[\overline{\mathtt{G}}](\overline{\mathtt{v}})\ \text{where } \overline{\mathtt{G}} \text{ contains no type variables}
\end{array}
$$

C, D range over class names, f, g over field names, m over method names, x, y over variable names, X, Y over type variables.

Figure 1. FXG productions.

$$\mathsf{fields}(\mathtt{Object}) = \bullet \qquad \text{(L-FIELDS-B)}$$

$$\frac{\mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{F}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{E}}]\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \qquad \theta = \overline{\mathtt{A}}/\overline{\mathtt{X}} \qquad \mathsf{fields}(\mathtt{D}[\overline{\mathtt{E}}\theta]) = \overline{\mathtt{g}}:\overline{\mathtt{G}} \qquad \overline{\mathtt{f}} \cap \overline{\mathtt{g}} = \emptyset}{\mathsf{fields}(\mathtt{C}[\overline{\mathtt{A}}]) = \overline{\mathtt{g}},\overline{\mathtt{f}}:\overline{\mathtt{G}},\overline{\mathtt{F}}\theta} \qquad \text{(L-FIELDS-I)}$$

$$\frac{\mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{F}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{E}}]\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \qquad \mathtt{def}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} = \mathtt{e} \in \overline{\mathtt{M}} \qquad \theta = \overline{\mathtt{A}},\overline{\mathtt{B}}/\overline{\mathtt{X}},\overline{\mathtt{Y}}}{\mathsf{methods}(\mathtt{C}[\overline{\mathtt{A}}]) \ni \mathtt{m}[\overline{\mathtt{B}}](\overline{\mathtt{x}}:\overline{\mathtt{G}\theta})\{\mathtt{d}\theta\}:\mathtt{H}\theta = \mathtt{e}\theta} \qquad \text{(L-METHOD-B)}$$

$$\frac{\mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{F}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{E}}]\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \qquad \theta = \overline{\mathtt{A}}/\overline{\mathtt{X}} \qquad \mathsf{methods}(\mathtt{D}[\overline{\mathtt{E}}\theta]) \ni \mathtt{m}[\overline{\mathtt{B}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} = \mathtt{e} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathsf{methods}(\mathtt{C}[\overline{\mathtt{A}}]) \ni \mathtt{m}[\overline{\mathtt{B}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} = \mathtt{e}} \qquad \text{(L-METHOD-I)}$$

Figure 2. FXG fields and methods.

$$\frac{\mathsf{fields}(\mathtt{C}[\overline{\mathtt{A}}]) = \overline{\mathtt{f}}:\overline{\mathtt{F}}}{\mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{v}}).\mathtt{f}_i \to \mathtt{v}_i} \quad \text{(R-FIELD)} \qquad \frac{\mathtt{e}_i \to \mathtt{e}_i'}{\mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\mathtt{v}_1,\ldots,\mathtt{v}_{i-1},\mathtt{e}_i,\ldots,\mathtt{e}_n) \to \mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\mathtt{v}_1,\ldots,\mathtt{v}_{i-1},\mathtt{e}_i',\ldots,\mathtt{e}_n)} \quad \text{(RC-NEW-ARG)}$$

$$\frac{\mathtt{e} \to \mathtt{e}'}{\mathtt{e.f} \to \mathtt{e}'.\mathtt{f}} \quad \text{(RC-FIELD)} \qquad \frac{\mathsf{methods}(\mathtt{C}[\overline{\mathtt{A}}]) \ni \mathtt{m}[\overline{\mathtt{B}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} = \mathtt{e} \qquad \theta = \mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{v}}),\overline{\mathtt{w}}/\mathtt{this},\overline{\mathtt{x}}}{\mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{v}}).\mathtt{m}[\overline{\mathtt{B}}](\overline{\mathtt{w}}) \to \mathtt{e}\theta} \quad \text{(R-INVK)}$$

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

$$\frac{\mathtt{e} \to \mathtt{e}'}{\mathtt{e}\ \mathtt{as}\ \mathtt{G} \to \mathtt{e}'\ \mathtt{as}\ \mathtt{G}} \quad \text{(RC-CAST)} \qquad \frac{\Gamma \vdash \overline{\mathtt{v}}:\overline{\mathtt{V}} \qquad \mathtt{x}:\exists\overline{\mathtt{y}}:\overline{\mathtt{V}}.\ \mathtt{C}[\overline{\mathtt{A}}]\{\mathtt{self} == \mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{y}})\} \vdash \mathtt{x} <: \mathtt{G}}{\mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{v}})\ \mathtt{as}\ \mathtt{G} \to \mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{v}})} \quad \text{(R-CAST)}$$

Figure 3. FXG operational semantics. $\overline{\mathtt{A}}$ and $\overline{\mathtt{B}}$ are lists of ground types (no type variables, no existentials).

$$\vdash_{\mathcal{W}} \mathtt{Object} \quad \text{(W-OBJECT)} \qquad \frac{\Gamma \vdash_{\mathcal{W}} \mathtt{T} \qquad \Gamma,\mathtt{self}:\mathtt{T} \vdash_{\mathcal{C}} \mathtt{c}}{\Gamma \vdash_{\mathcal{W}} \mathtt{T}\{\mathtt{c}\}} \quad \text{(W-DEP)} \qquad \frac{\Gamma \vdash_{\mathcal{W}} \mathtt{T} \qquad \Gamma,\mathtt{x}:\mathtt{T} \vdash_{\mathcal{W}} \mathtt{U}}{\Gamma \vdash_{\mathcal{W}} \exists \mathtt{x}:\mathtt{T}.\ \mathtt{U}} \quad \text{(W-EXISTS)}$$

$$\mathtt{X}:* \vdash_{\mathcal{W}} \mathtt{X} \quad \text{(W-TYPE)}$$

$$\frac{\mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{F}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{E}}]\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P} \qquad \Gamma \vdash_{\mathcal{W}} \overline{\mathtt{A}} \qquad \overline{\mathtt{X}}:* \vdash_{\mathcal{C}} \mathtt{c} \qquad \theta = \overline{\mathtt{A}}/\overline{\mathtt{X}} \qquad \sigma(\Gamma) \vdash_{\mathcal{X}} \mathtt{c}\theta}{\Gamma \vdash_{\mathcal{W}} \mathtt{C}[\overline{\mathtt{A}}]} \quad \text{(W-CLASS)}$$

Figure 4. FXG well-formedness.

$$\frac{\mathsf{fields}(\mathtt{C}[\overline{\mathtt{A}}]) = \overline{\mathtt{f}}:\overline{\mathtt{F}} \qquad \theta = \mathtt{x}/\mathtt{this}}{\Gamma,\mathtt{x}:\mathtt{C}[\overline{\mathtt{A}}] \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{f}_i:\mathtt{F}_i\theta} \quad \text{(H-FIELD)} \qquad \frac{\Gamma,\mathtt{x}:\mathtt{T},\mathtt{c} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{I}}{\Gamma,\mathtt{x}:\mathtt{T}\{\mathtt{c}\} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{I}} \quad \text{(H-DEP)}$$

$$\frac{\mathsf{methods}(\mathtt{C}[\overline{\mathtt{A}}]) \ni \mathtt{m}[\overline{\mathtt{B}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} = \mathtt{e} \qquad \theta = \mathtt{x},\overline{\mathtt{z}}/\mathtt{this},\overline{\mathtt{y}}}{\Gamma,\mathtt{x}:\mathtt{C}[\overline{\mathtt{A}}] \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{m}[\overline{\mathtt{B}}](\overline{\mathtt{z}}:\overline{\mathtt{G}\theta})\{\mathtt{d}\theta\}:\mathtt{H}\theta} \quad \text{(H-METHOD)} \qquad \frac{\Gamma,\mathtt{y}:\mathtt{T},\mathtt{x}:\mathtt{U} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{I}}{\Gamma,\mathtt{x}:\exists \mathtt{y}:\mathtt{T}.\ \mathtt{U} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{I}} \quad \text{(H-EXISTS)}$$

Figure 5. FXG member lookup. I ranges over members (fields and methods).

$$\frac{\mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{F}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{E}}]\ \{\ \overline{\mathtt{M}}\ \} \in \mathtt{P}\theta = \overline{\mathtt{A}}/\overline{\mathtt{X}}}{\Gamma, \mathtt{x}:\mathtt{C}[\overline{\mathtt{A}}] \vdash \mathtt{x} <: \mathtt{D}[\overline{\mathtt{E}}\theta]}\ \text{(S-CLASS)} \qquad \frac{\Gamma \vdash \mathtt{x} <: \mathtt{T} \qquad \Gamma, \mathtt{y}:\mathtt{T} \vdash \mathtt{y} <: \mathtt{U}}{\Gamma \vdash \mathtt{x} <: \mathtt{U}}\ \text{(S-TRANS)}$$

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

$$\frac{\Gamma, \mathtt{y}:\mathtt{U}, \mathtt{x}:\mathtt{S} \vdash \mathtt{x} <: \mathtt{T} \qquad \mathtt{y} \text{ not free in } \mathtt{T}}{\Gamma, \mathtt{x}:\exists \mathtt{y}:\mathtt{U}.\ \mathtt{S} \vdash \mathtt{x} <: \mathtt{T}}\ \text{(S-EXISTS-L)} \qquad \frac{\Gamma \vdash \mathtt{t}:\mathtt{U},\ \mathtt{y} <: \mathtt{T}[\mathtt{t}/\mathtt{x}]}{\Gamma \vdash \mathtt{y} <: \exists \mathtt{x}:\mathtt{U}.\ \mathtt{T}}\ \text{(S-EXISTS-R)}$$

Figure 6. FXG subtyping rules.

$$\Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x}:\mathtt{T}\{\mathtt{self} == \mathtt{x}\}\ \text{(T-VAR)} \qquad \frac{\Gamma \vdash \mathtt{e}:\mathtt{T}}{\Gamma \vdash \mathtt{e}\ \mathtt{as}\ \mathtt{G}:\mathtt{G}}\ \text{(T-CAST)} \qquad \frac{\Gamma \vdash \mathtt{e}:\mathtt{T} \qquad \mathtt{x} \text{ fresh} \qquad \Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{f}:\mathtt{F}}{\Gamma \vdash \mathtt{e.f}:\exists \mathtt{x}:\mathtt{T}.\ \mathtt{F}\{\mathtt{self} == \mathtt{x.f}\}}\ \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash \overline{\mathtt{e}}:\overline{\mathtt{T}} \qquad \mathsf{fields}(\mathtt{C}[\overline{\mathtt{A}}]) = \overline{\mathtt{f}}:\overline{\mathtt{F}} \qquad \mathtt{y}, \overline{\mathtt{x}} \text{ fresh} \qquad \theta = \mathtt{y}/\mathtt{this} \qquad \Gamma, \mathtt{y}:\mathtt{C}[\overline{\mathtt{A}}], \overline{\mathtt{x}}:\overline{\mathtt{T}}, \mathtt{y}.\overline{\mathtt{f}} == \overline{\mathtt{x}} \vdash \overline{\mathtt{x}} <: \overline{\mathtt{F}}\theta}{\Gamma \vdash \mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{e}}):\exists \overline{\mathtt{x}}:\overline{\mathtt{T}}.\ \mathtt{C}[\overline{\mathtt{A}}]\{\mathtt{self} == \mathtt{new}\ \mathtt{C}[\overline{\mathtt{A}}](\overline{\mathtt{x}})\}}\ \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{e}:\mathtt{T},\ \overline{\mathtt{a}}:\overline{\mathtt{U}} \qquad \mathtt{x}, \overline{\mathtt{y}} \text{ fresh} \qquad \Gamma, \mathtt{x}:\mathtt{T}, \overline{\mathtt{y}}:\mathtt{U} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{m}[\overline{\mathtt{A}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H},\ \mathtt{c},\ \overline{\mathtt{y}} <: \overline{\mathtt{G}}}{\Gamma \vdash \mathtt{e.m}[\overline{\mathtt{A}}](\overline{\mathtt{a}}):\exists \mathtt{x}:\mathtt{T}, \overline{\mathtt{y}}:\overline{\mathtt{U}}.\ \mathtt{H}}\ \text{(T-INVK)}$$

$$\Gamma \vdash \mathtt{f}:\mathtt{F} \ll \mathtt{f}:\mathtt{F}\ \text{(O-FIELD)} \qquad \frac{\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{G}}, \mathtt{d}' \vdash \mathtt{d} \qquad \mathtt{y} \text{ fresh} \qquad \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{G}}, \mathtt{d}, \mathtt{y}:\mathtt{H} \vdash \mathtt{y} <: \mathtt{H}'}{\Gamma \vdash \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} \ll \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}'\}:\mathtt{H}'}\ \text{(O-METHOD)}$$

$$\frac{\begin{array}{c}\mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{F}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{E}}]\ \{\ \overline{\mathtt{M}}\ \} \qquad \Gamma = \overline{\mathtt{X}}:*, \mathtt{c}, \mathtt{this}:\mathtt{C}[\overline{\mathtt{X}}], \overline{\mathtt{Y}}:* \\ \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{G}} \vdash_{\mathcal{C}} \mathtt{d} \qquad \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{G}}, \mathtt{d} \vdash_{\mathcal{W}} \overline{\mathtt{G}}, \mathtt{H} \qquad \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{G}}, \mathtt{d} \vdash \mathtt{e}:\mathtt{E} \qquad \mathtt{y} \text{ fresh} \qquad \Gamma, \overline{\mathtt{x}}:\overline{\mathtt{G}}, \mathtt{d}, \mathtt{y}:\mathtt{E} \vdash \mathtt{y} <: \mathtt{H} \\ \text{if } \overline{\mathtt{X}}:*, \mathtt{c}, \mathtt{this}:\mathtt{D}[\overline{\mathtt{E}}], \overline{\mathtt{Y}}:* \vdash \mathtt{this}\ \mathtt{has}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{x}}:\overline{\mathtt{G}'})\{\mathtt{d}'\}:\mathtt{H}' \text{ then } \Gamma \vdash \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} \ll \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{x}}:\overline{\mathtt{G}'})\{\mathtt{d}'\}:\mathtt{H}'\end{array}}{\vdash \mathtt{def}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{x}}:\overline{\mathtt{G}})\{\mathtt{d}\}:\mathtt{H} = \mathtt{e} \text{ OK in } \mathtt{C}[\overline{\mathtt{X}}]}\ \text{(OK-METHOD)}$$

$$\frac{\mathsf{fields}(\mathtt{D}[\overline{\mathtt{E}}]) = \overline{\mathtt{g}}:\overline{\mathtt{G}} \qquad \overline{\mathtt{f}} \cap \overline{\mathtt{g}} = \emptyset \qquad \overline{\mathtt{X}}:* \vdash_{\mathcal{C}} \mathtt{c} \qquad \overline{\mathtt{X}}:*, \mathtt{c} \vdash_{\mathcal{W}} \mathtt{D}[\overline{\mathtt{E}}] \qquad \overline{\mathtt{X}}:*, \mathtt{c}, \mathtt{this}:\mathtt{C}[\overline{\mathtt{X}}] \vdash_{\mathcal{W}} \overline{\mathtt{F}} \qquad \overline{\mathtt{M}} \text{ OK in } \mathtt{C}[\overline{\mathtt{X}}]}{\vdash \mathtt{class}\ \mathtt{C}[\overline{\mathtt{X}}]\{\mathtt{c}\}(\overline{\mathtt{f}}:\overline{\mathtt{F}})\ \mathtt{extends}\ \mathtt{D}[\overline{\mathtt{E}}]\ \{\ \overline{\mathtt{M}}\ \} \text{ OK}}\ \text{(OK-CLASS)}$$

Figure 7. FXG typing rules.

$$\begin{array}{rcll} \text{(Generic type)} & \mathtt{G} & ::= & \mathtt{R} \\ \text{(Expression)} & \mathtt{e} & ::= & \mathtt{q}(\overline{\mathtt{e}}) \\ \text{(Values)} & \mathtt{v} & ::= & \mathtt{l} \\ \text{(Constraint term)} & \mathtt{t} & ::= & \mathtt{q}(\overline{\mathtt{t}})\ \mid \mathtt{l} \\ \text{(Constraint)} & \mathtt{c} & ::= & \mathtt{p}(\overline{\mathtt{t}}) \end{array}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} \overline{\mathtt{t}} \qquad \mathtt{q} \in \mathcal{Q}}{\Gamma \vdash_{\mathcal{T}} \mathtt{q}(\overline{\mathtt{t}})}\ \text{(W-FUN)} \qquad \frac{\mathtt{l} \in \mathcal{L}}{\vdash_{\mathcal{T}} \mathtt{l}}\ \text{(W-LIT)} \qquad \frac{\vdash_{\mathcal{X}} \mathtt{q}(\overline{\mathtt{v}}) == \mathtt{l}}{\mathtt{q}(\overline{\mathtt{v}}) \rightarrow \mathtt{l}}\ \text{(R-FUN)}$$

$$\frac{\Gamma \vdash_{\mathcal{T}} \overline{\mathtt{t}} \qquad \mathtt{p} \in \mathcal{P}}{\Gamma \vdash_{\mathcal{C}} \mathtt{p}(\overline{\mathtt{t}})}\ \text{(W-PRED)} \qquad \frac{\mathtt{R} \in \mathcal{R}}{\vdash_{\mathcal{W}} \mathtt{R}}\ \text{(W-PRIM)} \qquad \vdash \mathtt{l}:\mathsf{Dom}(\mathtt{l})\{\mathtt{self} == \mathtt{l}\}\ \text{(T-LIT)}$$

$$\frac{\mathtt{e}_i \rightarrow \mathtt{e}_i'}{\mathtt{q}(\mathtt{v}_1, \ldots, \mathtt{v}_{i-1}, \mathtt{e}_i, \ldots, \mathtt{e}_n) \rightarrow \mathtt{q}(\mathtt{v}_1, \ldots, \mathtt{v}_{i-1}, \mathtt{e}_i', \ldots, \mathtt{e}_n)}\ \text{(RC-FUN)} \qquad \frac{\Gamma \vdash \overline{\mathtt{e}}:\mathsf{Dom}(\mathtt{q})}{\Gamma \vdash \mathtt{q}(\overline{\mathtt{e}}):\exists \overline{\mathtt{x}}:\mathsf{Dom}(\mathtt{q}).\ \mathsf{Rng}(\mathtt{q})\{\mathtt{self} == \mathtt{q}(\overline{\mathtt{x}})\}}\ \text{(T-FUN)}$$

Figure 8. FXG +primitive types.

$$\text{(Constraint)} \quad \mathtt{c} \quad ::= \quad \mathtt{X}\ \underline{\mathtt{has}}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H} \qquad \frac{\vdash \mathtt{X}\ \underline{\mathtt{has}}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H} \qquad \theta = \mathtt{x}, \overline{\mathtt{Z}}, \overline{\mathtt{z}}/\mathtt{this}, \overline{\mathtt{Y}}, \overline{\mathtt{y}}}{\Gamma, \mathtt{x}:\mathtt{X} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{m}[\overline{\mathtt{Z}}](\overline{\mathtt{z}}:\overline{\mathtt{G}\theta})\{\mathtt{c}\theta\}:\mathtt{H}\theta}\ \text{(H-STRUCT)}$$

$$\frac{\Gamma \vdash_{\mathcal{W}} \mathtt{X} \qquad \Gamma, \mathtt{this}:\mathtt{X}, \overline{\mathtt{Y}}:*, \overline{\mathtt{y}}:\overline{\mathtt{G}} \vdash_{\mathcal{C}} \mathtt{c} \qquad \Gamma, \mathtt{this}:\mathtt{X}, \overline{\mathtt{Y}}:*, \overline{\mathtt{y}}:\overline{\mathtt{G}}, \mathtt{c} \vdash_{\mathcal{W}} \overline{\mathtt{G}}, \overline{\mathtt{H}}}{\Gamma \vdash_{\mathcal{C}} \mathtt{X}\ \underline{\mathtt{has}}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H}}\ \text{(W-STRUCT)}$$

$$\frac{\mathsf{methods}(\mathtt{C}[\overline{\mathtt{A}}]) \ni \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H} = \mathtt{e}}{\vdash_{\mathcal{X}} \mathtt{C}[\overline{\mathtt{A}}]\ \underline{\mathtt{has}}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H}}\ \text{(X-STRUCT)} \qquad \frac{\begin{array}{c}\Gamma, \mathtt{x}:\mathtt{X} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H},\ \mathtt{x}\ \mathtt{has}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}'})\{\mathtt{c}'\}:\mathtt{H}' \\ \overline{\mathtt{G}} \neq \overline{\mathtt{G}'} \text{ or } \mathtt{H} \neq \mathtt{H}' \text{ or } \Gamma, \mathtt{x}:\mathtt{X}, \overline{\mathtt{y}}:\overline{\mathtt{G}}, \mathtt{c} \nvdash \mathtt{c}' \text{ or } \Gamma, \mathtt{x}:\mathtt{X}, \overline{\mathtt{y}}:\overline{\mathtt{G}'}, \mathtt{c}' \nvdash \mathtt{c}\end{array}}{\Gamma \vdash \mathtt{false}}\ \text{(KO-STRUCT)}$$

Figure 9. FXG +structural subtyping constraints.

will be checked to see if it is actually of type G (see R-CAST in Figure 3).

T-FIELD may be understood through "proxy" reasoning. Given the context Γ, assume the receiver e can be established to be of type T. Now, we do not know the run-time value of e, so we shall assume that it is some fixed but unknown "proxy" value x (of type T) that is "fresh" in that it is not known to be related to any known value (i.e., those recorded in Γ). If we can establish that x has a field f of type F then we can assert that e.f has type F and, further, that it equals x.f for some x of type T. Hence, we can assert that e.f has type $\exists \mathtt{x}:\mathtt{T}.\ \mathtt{F}\{\mathtt{self} == \mathtt{x.f}\}$.

T-NEW and T-INVK have a similar structure to T-FIELD: we use proxy reasoning for the arguments of the constructor call or for the receiver and the arguments of the method call. Both T-NEW and T-INVK check that the argument types are subtypes of the types of the formals. In addition, T-INVK requires the types of the receiver and the arguments to satisfy the method guard.

Fields cannot be overridden thanks to the premise $\bar{\mathtt{f}} \cap \bar{\mathtt{g}} = \emptyset$ in rule OK-CLASS. O-METHOD formalizes method overriding. A method $m1$ may be overridden by a method $m2$ with the same name, type parameters (modulo alpha-renaming), value parameters (modulo alpha-renaming), and value parameter types provided method $m2$ has a return type that is a subtype of $m1$'s return type and $m1$'s guard entails $m2$'s. Rule O-FIELD is redundant at this point. It will matter to type system extensions.

OK-METHOD and OK-CLASS ensure that all types and constraints are well-formed, that overriding rules are observed, and that the body of a method has a type that is a subtype of its declared type. Moreover, the guard on a class must entail the well-formedness of its supertype. which includes entailing the guard of the superclass.

3.2 FXG +primitive types

Since the FXG design is parametric in the constraint language and system 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 $\mathtt{R} \in \mathcal{R}$, predicates $\mathtt{p} \in \mathcal{P}$, functions $\mathtt{q} \in \mathcal{Q}$, and literals $\mathtt{l} \in \mathcal{L}$ of these primitive types. Second, we extend the productions, operational semantics and type system of FXG with the productions and inference rules of Figure 8.

We denote $\mathsf{Dom}(\mathtt{l})$ the primitive type of the literal l. We assume each function q is a total mapping from $\mathsf{Dom}(\mathtt{q})$ (an n-tuple of primitive types) to $\mathsf{Rng}(\mathtt{q})$, that is, if $\vdash \bar{\mathtt{v}} : \mathsf{Dom}(\mathtt{q})$ then there exists a unique literal l such that $\vdash_{\mathcal{X}} \mathtt{q}(\bar{\mathtt{v}}) == \mathtt{l}$ and moreover $\mathsf{Dom}(\mathtt{l}) = \mathsf{Rng}(\mathtt{q})$.

For instance, if $\mathcal{X}$ defines the type int, integer literals, and the usual arithmetic operators, we can declare:

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

3.3 Results

The following results hold for FXG +primitive types. Below, the typing context is always assumed to be consistent and well-formed.

THEOREM 3.1 (Principal types). *$\Gamma \vdash \mathtt{t}:\mathtt{T}$ and $\Gamma \vdash \mathtt{x}:\mathtt{U}$ then $\mathtt{T} = \mathtt{U}$.*

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

THEOREM 3.2 (Method guards). *If $\Gamma \vdash \mathtt{e.m}[\bar{\mathtt{B}}](\bar{\mathtt{a}}) : \mathtt{T}$ and $\mathtt{e} \rightarrow^* \mathtt{new\ C}[\bar{\mathtt{A}}](\bar{\mathtt{v}})$ and $\bar{\mathtt{a}} \rightarrow^* \bar{\mathtt{w}}$ and $\mathsf{methods}(\mathtt{C}[\bar{\mathtt{A}}]) \ni \mathtt{m}[\bar{\mathtt{B}}](\bar{\mathtt{x}}{:}\bar{\mathtt{G}})\{\mathtt{c}\}{:}\mathtt{H} = \mathtt{e}$ then $\Gamma \vdash \mathtt{c}[\mathtt{new\ C}[\bar{\mathtt{A}}](\bar{\mathtt{v}}), \bar{\mathtt{w}}/\mathtt{this}, \bar{\mathtt{x}}]$.*

LEMMA 3.3 (Subject Reduction). *If $\mathtt{e} \rightarrow \mathtt{e}'$, and $\Gamma \vdash \mathtt{e}:\mathtt{T}$ then there exists a type S such that $\Gamma \vdash \mathtt{e}':\mathtt{S}$. Moreover, $\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{x} <: \mathtt{T}$.*

LEMMA 3.4 (Progress). *If $\vdash \mathtt{e}:\mathtt{T}$ then one of the following conditions holds:*

1. *e is a value,*
2. *e contains a stuck cast sub-expression of the form "v as G",*
3. *there exists $\mathtt{e}'$ such that $\mathtt{e} \rightarrow \mathtt{e}'$.*

THEOREM 3.5 (Type soundness). *If $\vdash \mathtt{e}:\mathtt{T}$ and e reduces to a normal form $\mathtt{e}'$ then either $\mathtt{e}'$ contains a stuck cast sub-expression of the form "v as G" or $\mathtt{e}'$ is a value v and there exists S such that $\vdash \mathtt{v}:\mathtt{S}$. Moreover, in that case, $\mathtt{x}:\mathtt{S} \vdash \mathtt{x} <: \mathtt{T}$.*

The proof of these results is detailed in a technical report, which is available for download at x10.codehaus.org/Papers.

It relies on the following two pivotal lemmas:

LEMMA 3.6. *If $\Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x\ has\ I}, \mathtt{x\ has\ J}$ where I and J are both fields or both methods with the same name then $\mathtt{I} = \mathtt{J}$ modulo alpha-renaming of the type and value parameters.*

LEMMA 3.7. *If $\Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x} <: \mathtt{G}$ and $\Gamma, \mathtt{x}:\mathtt{G} \vdash \mathtt{x\ has\ I}$ then there exists J such that $\Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x\ has\ J}, \mathtt{J} \ll \mathtt{I}$.*

These lemmas makes it possible to separate the main body of the proof from the concrete treatment of type parameters. In the following section, we will further axiomatize type parameters. By making sure these language extensions preserve the two lemmas, we ensure the type system remain sound with minimal additions to the proof.

3.4 The FXG family

In the base FXG language, if a variable x has type-parameter type X then x has no accessible field or method. In this section, we demonstrate how to make type parameters more expressive while preserving principal and sound types.

The key idea is that information about type parameters can be accumulated through constraints. Then typing rules make use of these constraints to lookup members of variables of type-parameter types or establish subtyping relation about them.

Bounds. FGJ-style bounds on type parameters can be supported in the FXG family by introducing an "extends" constraint X <= G on type parameters.

We need to make several additions to FXG to make this work.

1. We extend the vocabulary of constraints:

$$\text{(Constraint)} \quad \mathtt{c} \quad ::= \quad \mathtt{X} \mathrel{<=} \mathtt{G}$$

2. We adopt a constraint system that can handle the new constraints and add inference rules about entailment if necessary—none in this case (see structural subtyping constraints below for an example). We also deal with inconsistent bounds, here by means of:

$$\frac{\mathtt{G} \neq \mathtt{H}}{\mathtt{X} \mathrel{<=} \mathtt{G},\ \mathtt{X} \mathrel{<=} \mathtt{H} \vdash_{\mathcal{X}} \mathtt{false}}$$

In this example, we only need to add entailment relations inside the constraint system. In general, we may want to derive false from the typing judgments as well.

3. We specify well-formedness conditions:

$$\frac{\Gamma \vdash_{\mathcal{W}} \mathtt{X} \qquad \Gamma \vdash_{\mathcal{W}} \mathtt{G}}{\Gamma \vdash_{\mathcal{C}} \mathtt{X} \mathrel{<=} \mathtt{G}}$$

4. We specify subtyping:

$$\frac{\Gamma \vdash_{\mathcal{C}} \mathtt{X} \mathrel{<=} \mathtt{G} \qquad \Gamma \vdash \mathtt{X} \mathrel{<=} \mathtt{G}}{\Gamma, \mathtt{x}:\mathtt{X} \vdash \mathtt{x} \mathrel{<:} \mathtt{G}}$$

5. We specify lookup:

$$\frac{\Gamma \vdash_{\mathcal{C}} \mathtt{X} \mathrel{<=} \mathtt{G} \qquad \Gamma \vdash \mathtt{X} \mathrel{<=} \mathtt{G}}{\Gamma, \mathtt{x}:\mathtt{X} \vdash \mathtt{x} \mathrel{<:} \mathtt{G}}$$

The bounds we have just specified are very elementary. The X10 type system supports lower and upper bounds as well as a much more permissive notion of compatible bounds (i.e., bounds are compatible if they have a common subtype). But the critical observation here is that a more elaborate extension would proceed from the exact same methodology: first add constraints, then subtyping rules and lookup rules, finally worry about inconsistent constraints.

Whatever the extension, to preserve the soundness of the type system and the property of principal types—that is, the proof structure we have previously described—one need only ensure that:

1. If a type parameter X is established to be a subtype of type G then each member I of G must be matched by a member J of X whose signature overrides that of I.

   Formally, if $\Gamma, \mathtt{x}:\mathtt{X} \vdash \mathtt{x} \mathrel{<:} \mathtt{G}$ and $\Gamma, \mathtt{x}:\mathtt{G} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{I}$ then there exists J such that $\Gamma, \mathtt{x}:\mathtt{X} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{J}, \mathtt{J} \ll \mathtt{I}$.

2. If a variable of type parameter X is established to have two members with the same name then they must have the same signature.

   Formally, if $\Gamma, \mathtt{x}:\mathtt{X} \vdash \mathtt{x}\ \mathtt{has}\ \mathtt{I}, \mathtt{x}\ \mathtt{has}\ \mathtt{J}$ where I and J are both fields or both methods with the same name then $\mathtt{I} = \mathtt{J}$ modulo alpha-renaming of the type and value parameters.

Since, the two properties holds for this extension, all results from Section 3.3 apply.

Structural subtyping constraints. To conclude, we illustrate this methodology with another language extension: structural subtyping constraints. For brevity, we only consider constraint $\mathtt{X}\ \underline{\mathtt{has}}\ \mathtt{m}[\overline{\mathtt{Y}}](\overline{\mathtt{y}}:\overline{\mathtt{G}})\{\mathtt{c}\}:\mathtt{H}$ and forget about fields. We use the underline notation "$\underline{\mathtt{has}}$" to distinguish the constraint on types from the predicate on variables. Of course, the two are closely related (by H-STRUCT).

This time, we actually need the constraint system to satisfy a new inference rule X-STRUCT so that generic classes and methods may be instantiated upon the actual class types of the program.

Rule KO-STRUCT precludes constraints inconsistent with requirements 1 and 2 above; all results from Section 3.3 apply.

4. 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 allows the design to capture many features of generics in object-oriented languages and then to extend these features with more expressive power. We have proved the type system sound.

We plan to extend the type system to account for mutable state. We believe the extension is straightforward, although cumbersome, because constraints are only immutable state only and because the formalism carefully controls occurrences of existential types.

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.

References

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

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 1.0, March 2008.

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

[4] Mark Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec♯ programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices, International Workshop, CASSIS 2004*, volume 2263 of *LNCS*. Springer-Verlag, 2004.

[5] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programing Language. In *OOPSLA*, 1998.

[6] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, Fall 1995.

[7] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *LNCS*, pages 27–51. Springer-Verlag, 1995.

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

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

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

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

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

[13] 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).

[14] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C♯ generics. In *Proc. 20th European Conference on Object-Oriented Programming (ECOOP)*, 2006.

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

[16] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, October 2002.

[17] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, Berlin, Germany, June 2002.

[18] Seth Fogarty, Emir Pašalić, Jeremy Siek, and Walid Taha. Concoq-

tion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 112–121, January 2007.

[19] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, June 1991.

[20] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *Proceedings of the 2007 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 133–152, 2007.

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

[22] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium*, 2007.

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

[24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1999.

[25] 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.

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

[27] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, Minneapolis, Minnesota, 2000.

[28] Xavier Leroy et al. The Objective Caml system. `http://caml.inria.fr/ocaml/`.

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

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

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

[32] 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.

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

[34] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP)*, number 5142 in Lecture Notes in Computer Science, July 2008.

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

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

[37] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[38] Todd Millstein. Practical predicate dispatch. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2004.

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

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

[41] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.

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

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

[44] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 2008 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2008.

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

[46] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

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

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

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

[50] Vijay Saraswat et al. The X10 language specification. Technical report, IBM T.J. Watson Research Center, 2008.

[51] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR*, 2005.

[52] 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.

[53] Don Syme and Andrew Kennedy. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

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

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

[56] 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.

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

[58] 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.

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