# Constrained Types for Object-Oriented Languages

Nathaniel Nystrom [*]
nystrom@us.ibm.com

Vijay Saraswat [*]
vsaraswa@us.ibm.com

Jens Palsberg [†]
palsberg@cs.ucla.edu

Christian Grothoff [‡]
christian@grothoff.org

## Abstract

X10 is a modern object-oriented language designed for productivity and performance in concurrent and distributed systems. In this setting, dependent types offer significant opportunities for detecting design errors statically, documenting design decisions, eliminating costly runtime checks (e.g., for array bounds, null values), and improving the quality of generated code.

We present the design and implementation of *constrained types*, a natural, simple, clean, and expressive extension to object-oriented programming: A type `C{c}` names a class or interface `C` and a *constraint* `c` on the immutable state of `C` and in-scope final variables. Constraints may also be associated with class definitions (representing class invariants) and with method and constructor definitions (representing preconditions). Dynamic casting is permitted. The system is parametric on the underlying constraint system: the compiler supports a simple equality-based constraint system but, in addition, supports extension with new constraint systems using compiler plugins.

***Categories and Subject Descriptors*** D.3.2 [*Language Classifications*]: Object-oriented languages; D.3.3 [*Language Constructs and Features*]: Classes and objects, Constraints

***General Terms*** Languages

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

[†] UCLA Computer Science Department, Boelter Hall, Los Angeles CA 90095 USA

[‡] Department of Computer Science, University of Denver, 2360 S. Gaylord Street, John Green Hall, Room 214, Denver CO, 80208 USA



*OOPSLA'08,* October 19–23, 2008, Nashville, Tennessee, USA.


## 1. Introduction

X10 is a modern statically typed object-oriented language designed for high productivity in the high performance computing (HPC) domain [57]. Built essentially on sequential imperative object-oriented core similar to Scala or Java™, X10 introduces constructs for distribution and fine-grained concurrency (asynchrony, atomicity, ordering).

The design of X10 requires a rich type system to permit a large variety of errors to be ruled out at compile time and to generate efficient code. X10, like most object-oriented languages supports classes; however, it places equal emphasis on *arrays*, a central data structure in high performance computing. In particular, X10 supports dense, distributed multi-dimensional arrays of value and reference types, built over index sets known as *regions*.

A key goal of X10 is to rule out large classes of error by design. For instance, the possibility of indexing a 2-d array with 3-d points should simply be ruled out at compile-time. This means that one must permit the programmer to express types such as `Region(2)`, the type of all two-dimensional regions; `Array[int](5)`, the type of all arrays of `int` of length `5`; `Array[int](Region(2))`, the type of all `int` arrays over two-dimensional regions; and `Tree{loc==here}`, the type of all `Tree` objects located on the current node. For concurrent computations, one needs the ability to statically check that a method is being invoked by an activity that is registered with a given clock (i.e., dynamic barrier) [57].

For performance, it is necessary that array index accesses are bounds-checked statically. Further, certain regions (e.g., rectangular regions) may be represented particularly efficiently. Hence, if a variable is to range only over rectangular regions, it is important that this information is conveyed through the type system to the code generator.

In this paper we describe X10's support for *constrained types*, a form of *dependent type* [37, 62, 48, 5, 6, 3, 16]—types parametrized by values—defined on predicates over the *immutable* state of objects. Constrained types statically capture many common invariants that naturally arise in code. For instance, typically the shape of an array (the number of

dimensions (the rank) and the size of each dimension) is determined at run time, but is fixed once the array is constructed. Thus, the shape of an array is part of its immutable state. Both mutable and immutable variables may have a constrained type: the constraint specifies invariants on the immutable state of the object stored in the variable.

X10 provides a framework for specifying and checking constrained types that achieves certain desirable properties:

- **Ease of use.** The syntax of constrained types is a simple and natural extension of nominal class types.
- **Flexibility.** The framework permits the development of concrete, specific type systems tailored to the application area at hand. X10's compiler permits extension with different constraint systems via compiler plugins, enabling a kind of pluggable type system [9]. The framework is parametric in the kinds of expressions used in the type system, permitting the installed constraint system to interpret the constraints.
- **Modularity.** The rules for type-checking are specified once in a way that is independent of the particular vocabulary of operations used in the dependent type system. The type system supports separate compilation.
- **Static checking.** The framework permits mostly static type-checking. The user is able to escape the confines of static type-checking using dynamic casts.

### 1.1 Constrained types

X10's sequential syntax is similar to Scala's. We permit the definition of a class `C` to specify a list of typed parameters or *properties*, $\mathtt{f}_1 : \mathtt{T}_1, \ldots, \mathtt{f}_k : \mathtt{T}_k$, similar in syntactic structure to a method formal parameter list. Each property in this list is treated as a public final instance field. We also permit the specification of a *class invariant* in the class definition. A class invariant is a boolean expression on the properties of the class. The compiler ensures that all instances of the class created at run time satisfy the invariant. Syntactically, the class invariant follows the property list. For instance, we may specify a class `List` with an `int length` property as follows:

```
class List(length: int){length >= 0} {...}
```

Given such a definition for a class `C`, types can be constructed by *constraining* the properties of `C`. In principle, *any* boolean expression over the properties specifies a type: the type of all instances of the class satisfying the boolean expression. Thus, `List{length == 3}` is a permissible type, as are `List{length <= 42}` and even `List{length * f() >= g()}` where `f` and `g` are functions on the immutable state of the `List` object and the variables in scope where the type appears. In practice, the constraint expression is restricted by the particular constraint system in use.

```
1  class List(n: int{self >= 0}) {
2    var head: Object = null;
3    var tail: List(n-1) = null;
4
5    def this(): List(0) { property(0); }
6
7    def this(head: Object, tail: List): List(tail.n+1) {
8      property(tail.n+1);
9      this.head = head;
10     this.tail = tail;
11   }
12
13   def append(arg: List): List(n+arg.n) {
14     return n==0
15       ? arg : new List(head, tail.append(arg));
16   }
17
18   def reverse(): List(n) = rev(new List());
19   def rev(acc: List): List(n+acc.n) {
20     return n==0
21       ? acc : tail.rev(new List(head, acc));
22   }
23
24   def filter(f: Predicate): List{self.n <= this.n} {
25     if (n==0) return this;
26     val l: List{self.n <= this.n-1} = tail.filter(f);
27     return (f.isTrue(head)) ? new List(head,l) : l;
28   }
29 }
```

**Figure 1.** This program implements a mutable list of Objects. The size of a list does not change through its lifetime, even though at different points in time its head and tail might point to different structures.

Our basic approach to introducing constrained types into X10 is to follow the spirit of generic types, but to use values instead of types.

In general, a *constrained type* is of the form `C{e}`, the name of a class or interface[1] `C`, called the *base class*, followed by a *condition* `e`, a boolean expression on the properties of the base class and the final variables in scope at the type. Such a type represents a refinement of `C`: the set of all instances of `C` whose immutable state satisfies the condition `e`. We write `C` for the vacuously constrained type `C{true}`, and write $\mathtt{C}(\mathtt{e}_1, \ldots, \mathtt{e}_k)$ for the type $\mathtt{C}\{\mathtt{f}_1{==}\mathtt{e}_1, \ldots, \mathtt{f}_k{==}\mathtt{e}_k\}$ where `C` declares the $k$ properties $\mathtt{f}_1, \ldots, \mathtt{f}_k$.

Constrained types may occur wherever normal types occur. In particular, they may be used to specify the types of properties, (possibly mutable) local variables or fields, arguments to methods, return types of methods; they may also be used in casts, etc.

Using the definitions above, `List(n)`, shown in Figure 1, is the type of all lists of length `n`. Intuitively, this definition states that a `List` has an `int` property `n`, which must be non-negative. The properties of the class are set through the in-

[1] In X10, primitive types such as `int` and `double` are object types; thus, for example, `int{self==0}` is a legal constrained type.

vocation of `property(...)` (analogously to `super(...)`) in the constructors of the class.

In a constraint, the name `self` is bound and refers to the type being constrained. The name `this`, by contrast, is a free variable in the constraint and refers to the receiver parameter of the current method or constructor. Use of `this` is not permitted in static methods.

The `List` class has two fields that hold the head and tail of the list. The fields are declared with the `var` keyword, indicating that they are not final. Variables declared with the `val` keyword, or without a keyword are final.

Constructors have "return types" that can specify an invariant satisfied by the object being constructed. The compiler verifies that the constructor return type and the class invariant are implied by the `property` statement and any `super` calls in the constructor body. A constructor must either invoke another constructor of the same class via a `this` call or must have a `property` statement on every non-exceptional path to ensure the properties are initialized. The `List` class has two constructors: the first constructor returns an empty list; the second returns a list of length `m+1`, where `m` is the length of the second argument.

In the second constructor (lines 7–11), as well as the `append` (line 13) and `rev` (line 20) methods, the return type depends on properties of the formal parameters. If an argument appears in a return type then the parameter must be final, ensuring the argument points to the same object throughout the evaluation of the method or constructor body. A parameter may also depend on another parameter in the argument list.

The use of constraints makes existential types very natural. Consider the return type of `filter` (line 24): it specifies that the list returned is of some unknown length. The only thing known about it is that its size is bounded by `n`. Thus, constrained types naturally subsume existential dependent types. Indeed, every base type `C` is an "existential" constrained type since it does not specify any constraint on its properties. Thus, code written with constrained types can interact seamlessly with legacy library code—using just base types wherever appropriate.

The return type of `filter` also illustrates the difference between `self` and `this`. Here, `self` refers to the `List` being returned by the method; `this` refers to the method's receiver.

### 1.2 Constraint system plugins

The X10 compiler allows programmers to extend the semantics of the language with compiler plugins. Plugins may be used to support different constraint systems to be used in constrained types. Constraint systems provide code for checking consistency and entailment.

The condition of a constrained type is parsed and type-checked as a normal boolean expression over properties and the `final` variables in scope at the type. Installed constraint systems translate the expression into an internal form, rejecting expressions that cannot be represented. A given condition may be a conjunction of constraints from multiple constraint systems. A Nelson–Oppen procedure [42] is used to check consistency of the constraints.

The X10 compiler implements a simple equality-based constraint system. Constraint solver plugins have been implemented for inequality constraints, for Presburger constraints using the CVC3 theorem prover [8], and for set-based constraints also using CVC3. These constraint systems are described in Section 3 and the implementation is discussed in Section 4.

### 1.3 Claims

The paper presents constrained types in the X10 programming language. We claim that the design is natural, easy to use, and useful. Many example programs have been written using constrained types and are available at `x10.sf.net/applications/examples`.

As in staged languages [43, 59], the design distinguishes between compile-time and run-time evaluation. Constrained types are checked (mostly) at compile-time. The compiler uses a constraint solver to perform universal reasoning (e.g., "for all possible values of method parameters") for dependent type-checking. There is no run-time constraint-solving. However, run-time casts and `instanceof` checks involving dependent types are permitted; these tests involve arithmetic, not algebra—the values of all parameters are known.

The design supports separate compilation: a class needs to be recompiled only when it is modified or when the method and field signatures or invariants of classes on which it depends are modified.

We claim that the design is flexible. The language design is parametric on the constraint system being used. The compiler supports integration of different constraint solvers into the language. Dependent clauses also form the basis of a general user-definable annotation framework we have implemented separately [46].

We claim the design is clean and modular. We present a simple core language CFJ, extending FJ [29] with constrained types on top of an arbitrary constraint system. We present rules for type-checking CFJ programs that are parametric in the constraint system and establish subject reduction and progress theorems.

***Rest of this paper.*** Section 2 describes the syntax and semantics of constrained types. Section 3 works through a number of examples using a variety of constraint systems. The compiler implementation, including support for constraint system plugins, is described Section 4. A formal semantics for a core language with constrained types is presented in Section 5, and a soundness proof is presented in the appendix. Section 6 reviews related work. The paper concludes in Section 7 with a discussion of future work.

## 2. Constrained types

This section describes constrained types in X10.

### 2.1 Properties

A property is a `public` final instance field of the class that cannot be overridden by subclassing. Like any other field, a property is typed, and its type need not necessarily be primitive. Thus, properties capture the immutable public state of an object, initialized when the object is created, that can be classified by constrained types. Syntactically, properties are specified in a parameter list right after the name of the class in a class definition. The class body may contain specifications of other fields; these fields are considered mutable.

Properties may be of arbitrary type. For instance, the class `region` has an `int` property called `rank`. In turn, the class `dist` has a `region` property, called `region`, and also an `int` property `rank`. The invariant for `dist` ensures that `rank == region.rank`. Similarly, an array has properties `dist`, `region`, and `rank` and appropriate constraints ensuring that the statically available information about them is consistent.[2] In this way, rich constraints on the immutable portion of the object reference graph, rooted at the current object and utilizing objects at user-defined types, may be specified.

### 2.2 Constraints

A constrained type is of the form `C{e}`, consisting of a *base class* `C` and a *condition* `e`, a boolean expression on the properties of the base class and the final variables in scope at the type. Constraints specify (possibly) partial information about the variables of interest. The type `C{e}` represents the set of all instances of `C` whose immutable state satisfies the condition `e`.

Constraints may use the special variable `self` to stand for the object whose type is being defined. Thus, `int{self>=0}` is the set of natural numbers, and `Point{x*x+y*y <= 1.0}` represents the interior of a circle (for a class `Point` with two `float` properties `x` and `y`). The type `C{self != null}` represents all instances of `C`. When there is no ambiguity, a property reference `self.x` may be abbreviated to `x`. The type `int{self==v}` represents a "singleton" type, an `int` is of this type only if it has the same value as `v`.

To be clear, `self` is not the same as `this`. In the `List` example of Figure 1, the `filter` method (line 24) returns a list with type `List{self.n <= this.n}`; the term `self.n` is the length of the returned `List`; the term `this.n` is the length of the receiver of the call to `filter`.

Constraints are specified in terms of an underlying constraint system [56]—a pre-defined logical vocabulary of functions and predicates with algorithms for consistency and entailment. The X10 compiler permits different constraint systems to be installed using compiler plugins [9]. Constraint system plugins define a language of constraints by symbolically interpreting the boolean expression specifying a type's condition; plugins may report an error if the condition cannot be interpreted.

In principle, types may be constrained by any boolean expression over the properties. For practical reasons, restrictions need to be imposed to ensure constraint checking is decidable.

The condition of a constrained type must be a pure function only of the properties of the base class. Because properties are final instance fields of the object, this requirement ensures that whether or not an object belongs to a constrained type does not depend on the *mutable* state of the object. That is, the status of the predicate "this object belongs to this type" does not change over the lifetime of the object. Second, by insisting that each property be a *field* of the object, the question of whether an object is of a given type can be determined merely by examining the state of the object and evaluating a boolean expression. Of course, an implementation is free to not *explicitly* allocate memory in the object for such fields. For instance, it may use some scheme of tagged pointers to implicitly encode the values of these fields.

Further, by requiring that the programmer distinguish certain final fields of a class as properties, we ensure that the programmer consciously controls *which* final fields should be available for constructing constrained types. A field that is "accidentally" final may not be used in the construction of a constrained type. It must be declared as a property.

### 2.3 Subtyping

Constrained types come equipped with a subtype relation that combines the nominal subtyping relation of classes and interfaces with the logical entailment relation of the constraint system. Namely, a constraint `C{c}` is a subtype of `D{d}` if `C` is a subtype of `D` and every value in `C` that satisfies `c` also satisfies `d`.

This definition implies that $\texttt{C\{e}_1\texttt{\}}$ is a subtype of $\texttt{C\{e}_2\texttt{\}}$ if $\texttt{e}_1$ implies $\texttt{e}_2$. In particular, for all conditions `e`, `C{e}` is a subtype of `C`. `C{e}` is empty exactly when `e` conjoined with `C`'s class invariant is inconsistent.

Two constrained types $\texttt{C}_1\texttt{\{e}_1\texttt{\}}$ and $\texttt{C}_2\texttt{\{e}_2\texttt{\}}$ are considered equivalent if $\texttt{C}_1$ and $\texttt{C}_2$ are the same base type and $\texttt{e}_1$ and $\texttt{e}_2$ are equivalent when considered as logical expressions. Thus, `C{x*x==4}` and `C{x==2 || x==-2}` are equivalent types.

### 2.4 Final variables

The use of final local variables, formal parameters, and fields in constrained types has proven to be particularly valuable in practice. The same variable that is being used in computation can also be used to specify types. There is no need to introduce separate, universally and existentially quantified "index" variables as in, for instance, DML [62]. During type-checking, final variables are turned into symbolic

[2] All constraint languages used in constrained types permit object references, field selection and equality. Such constraint systems have been studied extensively under the name of "feature structures" [2].

variables—some fixed but unknown value—of the same type. Computation is performed in a constraint-based fashion on such variables.

Because of the usefulness of final variables in X10, variables and parameters declared without an explicit `var` or `val` keyword are considered final.

### 2.5 Method and constructor preconditions

Methods and constructors may specify constraints on their parameters, including the implicit parameter `this`. For an invocation of a method (or constructor) to be type-correct, the associated constraint must be statically known to be satisfied by the actual receiver and actual arguments of the invocation. The constraint thus imposes a *precondition* on callers of the method.

The return type of a method may also contain expressions involving the arguments to the method. Any argument used in this way must be declared final, ensuring it is not mutated by the method body. For instance, the following is a valid method declaration in a class `Region` with a `boolean` property `rect`:

```
def product(r: Region{rect}){this.rect}:
        Region{rect} = ...
```

The method is intended to compute the Cartesian product of two rectangular regions (sets of $n$-dimensional points). The precondition `this.rect` says that the receiver must be rectangular for the method to be invoked.

### 2.6 Method overloading and overriding

The definitions of method overloading, overriding, hiding, shadowing and obscuring in X10 are the same as in Java [26], modulo the following considerations motivated by dependent types.

Our current implementation erases dependent type information when compiling to Java. Therefore it must be the case that a class does not have two different method definitions that conflict with each other when the constrained clauses in their types are erased.

A class `C` inherits from its direct superclass and superinterfaces all their methods that are visible according to the access modifiers and that are not hidden or overridden. A method `m1` in a class `C` overrides a method `m2` in a superclass `D` if `m1` and `m2` have signatures with equivalent (unerased) formal parameter types. It is a static error if `m1`'s erased signature is the same as `m2`'s, but `m1` does not override `m2`.

It is also a static error if the method precondition on `m2` does not entail the precondition on `m1`. This restriction ensures that if the a call is type-checked against `m2` at the superclass type `D`, the precondition is satisfied if the method is dispatched to the method `m1` in the subclass `C`.

Method dispatch takes only the class hierarchy into account, not dependent type information. Thus, X10 does not provide a form of predicate dispatch [11, 39], evaluating constraints at run time to determine which method to invoke. This design decision ensures that serious errors such as method invocation errors are captured at compile time. Such errors can arise because multiple incomparable methods with the same name and acceptable argument lists might be available at the dynamic dependent type of the subject.

### 2.7 Constructors for dependent classes

Constructors must ensure that the class invariants of the given class and its superclasses and superinterfaces hold. For instance, the nullary constructor for `List` ensures that the property `length` has the value `0`:

```
public def this(): List(0) { property(0); }
```

The `property` statement is used to set all the properties of the new object simultaneously. Capturing this assignment in a single statement simplifies checking that the constructor postcondition and class invariant are established. If a class has properties, every path through the constructor must contain exactly one `property` statement.

Java-like languages permit constructors to throw exceptions. This is necessary to deal with the situation in which the arguments to a constructor for a class `C` are such that no object can be constructed that satisfies the invariants for `C`. Dependent types make it possible to perform some of these checks at compile time. The class invariant of a class explicitly captures conditions on the properties of the class that must be satisfied by any instance of the class. Constructor preconditions capture conditions on the constructor arguments. The compiler's static check for non-emptiness of the type of any variable captures these invariant violations at compile time.

The class invariant is part of the public interface of the class. Consequently, if the invariant of `C` is changed, a class that creates instances of `C` may need to be recompiled to ensure the invariant is satisfied by the instances.

### 2.8 Extending dependent classes

A class may extend a constrained class, e.g., `class C(...) extends D{d}`. This documents the programmer's intention that every call to `super` in a constructor for `C` must ensure that the invariant `d` is established on the state of the class `D`. The expressions in the actual parameter list for the super class may involve only the properties of the class being defined.

### 2.9 Dependent interfaces

Java does not allow interfaces to specify instance fields. Rather, all fields in an interface are final static fields (constants). However, in X10 since properties play a central role in the specification of refinements of a type, it makes sense to permit interfaces to specify properties. Similarly, an interface definition may specify an invariant on its properties: all classes implementing the interface must satisfy the invariant. Methods in the body of an interface may have constraints on their parameters as well.

All classes implementing an interface must have a property with the same name and type (either declared in the class or inherited from the superclass) for each property in the interface. If a class implements multiple interfaces and more than one of them specifies a property with the same name, then they must all agree on the type of the property. The class must declare a single property with the given name and type.

Further, every method specified in the interface must have a corresponding method in the class with the same signature whose precondition, if any, is implied by the precondition of the method in the interface.

The general form of a class declaration is thus:

```
class C(x1: C1{c1}, ..., xk: Ck{ck}){c}
  extends D{d}
  implements I1{c1}, ..., In{cn} {...}
```

For all instances of `C`, the class invariant `c`, the invariants of the superclass and superinterfaces, as well as the constraints `d` and $c_i$ must hold.

### 2.10 Separation between compile-time and run-time computation

Our design distinguishes between compile-time execution (performed during type-checking) and run-time execution. At compile time, the compiler processes the abstract syntax tree of the program generating queries to the constraint solver. The only computation engine running is the constraint solver, which operates on its own vocabulary of predicates and functions. Program variables (such as local variables) that occur in types are dealt with symbolically. They are replaced with logical variables—some fixed, but unknown value—of the same type. The constraint solver must know how to process pieces of partial information about these logical variables in order to determine whether some constraint is entailed. At run time, the same program variable will have a concrete value and will perform "arithmetic" (calculations) where the compiler performed "algebra" (symbolic analysis).

Constrained types may occur in a run-time cast `e as T`. Code is generated to check at run time that the expression `e` satisfies any constraints in `T`.

### 2.11 Equality-based constraints

The X10 compiler includes a simple equality-based constraint system. All constraint systems installed using plugins must support at least the core equality-based constraints. Constraints are conjunctions of equalities between constraint terms: properties, final variables, compile-time constants, and `self`:

```
(C Term)     t   ::=  x | self | this | t.f | n
(Constraint) c,d ::=  true | t==t | c & c
                  |   x : T; c
```

We use the syntax `x: T; c` for the constraint obtained by existentially quantifying the variable `x` of type `T` in `c`.

### 2.12 Existential quantification

Constrained types subsume existential types. For example, the length of the list returned by `filter` in Figure 1 is existentially quantified.

Operations on values of constrained type propagate constraints to the operation result by introducing existentially quantified variables. Consider the assignment to `c` below:

```
a: int{self >= 0} = ...;
b: int{self >= 0} = ...;
c: int{self >= 0} = a*b;
```

During type-checking, the type of `a*b` is computed from the types of `a` and `b` to be the type:

```
int{x: int, y: int; self==x*y & x>=0 & y>=0)
```

That is, there exist non-negative `int`s `x` and `y` whose product is `self`. The constraint on this type is strong enough to establish the constraint on the return type. If the computed constraint cannot be represented by any installed constraint system, the type of `a*b` is promoted to the unconstrained supertype `int`.

### 2.13 Real clauses

Because object-oriented languages permit arbitrary mutual recursion between classes: classes `A` and `B` may have fields of type `B` and `A`, respectively—the type/property graph may have cycles. The nodes in this graph are base types (class and interface names). There is an edge from node `A` to node `B` if `A` has a property whose base type is `B`.

Let us define the *real clause* of a constrained type `C{c}` to be the set of constraints that must be satisfied by any instance of `C{c}`. This includes not only the condition `c` but also constraints that hold for all instances of `C`, as determined by `C`'s class invariant. Let $rc(\texttt{C\{c\}})$ denote the *real clause* of `C{c}`. For simplicity, we consider only top-level classes; thus, the only free variable in $rc(\texttt{C\{c\}})$ is `self`. We draw out `self` as a formal parameter and write $rc(\texttt{C\{c\}},\texttt{z})$ for $rc(\texttt{C\{c[z/self]\}})$.

Consider a general class definition:

```
class C(x1: C1{c1}, ..., xk: Ck{ck}){c}
  extends D{d} { ... }
```

From this, we get:

$$rc(\texttt{C},\texttt{z}) \iff \begin{array}{l}(\texttt{c}\wedge\texttt{d})[\texttt{z}/\texttt{self},\texttt{z}/\texttt{this}] \wedge rc(\texttt{D},\texttt{z}) \wedge \\ rc(\texttt{C}_1\{\texttt{c}_1\},\texttt{z.x}_1) \wedge \cdots \wedge rc(\texttt{C}_k\{\texttt{c}_k\},\texttt{z.x}_k)\end{array}$$

That is, given a program $P$ with classes $\texttt{C}_1,\ldots,\texttt{C}_k$, the set of real clauses for $\texttt{C}_1,\ldots,\texttt{C}_k$ are defined in a mutually recursive fashion through the Clark completion of a Horn clause theory (over an underlying constraint system).

The central algorithmic question now becomes whether given a constrained clause `d`, does $rc(\texttt{C\{c\}},\texttt{z})$ entail `d`? From

the above formulation the question is clearly semi-decidable. It is not clear however whether it is decidable. This is a direction for further work.

The X10 compiler is conservative and rejects programs with cyclic real clauses: programs where the real clause of the type of a property `p` itself constrains `p`. In practice, many data structures have non-cyclic real clauses. For these programs, the real clause can be computed quickly and only a bounded number of questions to the constraint solver are generated during type-checking.

### 2.14 Parametric consistency

Consider the set of final variables that are referenced in a type `T = C{c}`. These are the *parameters* of the type. A type is said to be *parametrically consistent* if its (class) invariant `c` is solvable for each possible assignment of values to parameters. Types are required to be parametrically consistent. Parametric consistency is sufficient to ensure that the extension of a type is non-empty.[3]

Consider a variation of `List` from Figure 1:

```
class List(n: int{self >= 0}) {
  var head: Object;
  var tail: List{self!=null &
                 self.n==this.n-1};
  ...
}
```

The type of the field `tail` is not parametrically consistent. There exists a value for the property `this.n`, namely `0`, for which the real clause `self != null & self.n == this.n-1 & self.n >= 0` is not satisfiable. Permitting `tail` to be `null` would allow the type to be non-empty.

The compiler will throw a type error when it encounters the initializer for this field in a constructor since it will not be able to prove that the initial value is of the given type.

## 3. Examples

The following section presents example uses of constrained types using several different constraint systems.

### 3.1 Equality constraints

The X10 compiler includes a simple equality-based constraint system, described in Section 2. Equalities constraints are used throughout X10 programs. For example, to ensure *n*-dimensional arrays are indexed only be *n*-dimensional index points, the array access operation requires that the array's `rank` property be equal to the index's `rank`.

Equality constraints specified in the X10 run-time library are used by the compiler to generate efficient code. For instance, an iteration over the points in a region can be optimized to a set of nested loops if the constraint on the region's type specifies that the region is rectangular and of constant rank.

### 3.2 Presburger constraints

Presburger constraints are linear integer inequalities. A Presburger constraint solver plugin was implemented using CVC3 [7, 8]. The list example in Figure 1 type-checks using this constraint system.

Presburger constraints are particularly useful in a high-performance computing setting where array operations are pervasive. Xi and Pfenning proposed using dependent types for eliminating array bounds checks [61]. A Presburger constraint system can be used to keep track of array dimensions and array indices to ensure bounds violations do not occur.

### 3.3 Set constraints: region-based arrays

Rather than using Presburger constraints, X10 takes another approach: following ZPL [10], arrays in X10 are defined over *regions*, sets of *n*-dimensional *index points* [27]. For instance, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`.

Regions and points were modeled in CVC3 [8] to create a constraint solver that ensures array bounds violations do not occur: an array access type-checks if the index point can be statically determined to be in the region over which the array is defined.

Region constraints are subset constraints written as calls to the `contains` method of the `region` class. The constraint solver does not actually evaluate the calls to the `contains` method, rather it interprets these calls symbolically as subset constraints at compile time.

Constraints have the following syntax:

| | | | |
|---|---|---|---|
| (Constraint) | `c` | ::= | `r.contains(r)` \| ... |
| (Region) | `r` | ::= | `t` \| $[b_1{:}d_1,\ldots,b_k{:}d_k]$ |
| | | \| | `r` \| `r` \| `r & r` \| `r - r` |
| | | \| | `r + p` \| `r - p` |
| (Point) | `p` | ::= | `t` \| $[b_1,\ldots,b_k]$ |
| (Integer) | `b,d` | ::= | `t` \| `n` |

where `t` are constraint terms (properties and final variables) and `n` are integer literals.

Regions used in constraints are either constraint terms `t`, region constants, unions (`|`), intersections (`&`), or differences (`-`), or regions where each point is offset by another point `p` using + or -.

For example, the code in Figure 2 performs a successive over-relaxation [53] of a matrix `G` with rank 2. The function declares a region variable `outer` as an alias for `G`'s region and a region variable `inner` to be the subset of `outer` that excludes the boundary points, formed by intersecting the `outer` region with itself shifted up, down, left, and right by one. The function then declares two more regions `d0` and `d1`, where $\mathtt{d}_i$ is set of points $\mathtt{x}_i$ where $(\mathtt{x}_0, \mathtt{x}_1)$ is in `inner`.

[3] Parametric consistency is not necessary in that there may be programs whose types are parametrically inconsistent but which never encounter empty types at run time because of relationships in data values that are too complicated to be captured by the type system.

```
1  const NORTH: point{rank==2} = [1,0];
2  const WEST:  point{rank==2} = [0,1];
3
4  def sor(omega: double,
5          G: Array[double]{rank==2},
6          iter: int): void {
7    outer: region{self==G.region & rank==2} = G.region;
8    inner: region{G.region.contains(self) &
9                  rank==G.region.rank}
10           = outer & (outer-NORTH) & (outer+NORTH)
11                   & (outer-WEST)  & (outer+WEST);
12
13   d0: region = inner.rank(0); // {i | (i,j) in inner}
14   d1: region = inner.rank(1); // {j | (i,j) in inner}
15
16   if (d1.size() == 0) return;
17
18   d1min: int = d1.low();
19   d1max: int = d1.high();
20
21   for (var off: int = 1; off <= iter*2; off++)
22     finish foreach ((i): point in d0)
23       if (i % 2 == off % 2)
24         for (ij: point in inner & [i..i,d1min..d1max])
25           G(ij) = omega / 4.
26                 * (G(ij-NORTH) + G(ij+NORTH)
27                  + G(ij-WEST)  + G(ij+WEST))
28                 * (1. - omega) * G(ij);
29 }
```

**Figure 2.** Successive over-relaxation with regions

The function iterates multiple times over points `i` in `d0`. The syntax `finish foreach` (line 22) tells the compiler to execute each loop iteration in parallel and to wait for all concurrent activities to terminate. The inner loop (lines 24–28) iterates over a subregion of `inner`.

The type checker establishes that the `region` property of the point `ij` (line 24) is `inner & [i..i,d1min..d1max]`, and that this region is a subset of `inner`, which is in turn a subset of `outer`, the region of the array `G`. Thus, the accesses to the array in the loop body do not violate the bounds of the array.

A key to making the program type-check is that the region intersection that defines `inner` (lines 10–11) is explicitly intersected with `outer` so that the constraint solver can determine that the result is a subset of `outer`.

## 4. Implementation

The X10 compiler provides a framework for writing and checking constrained types. Constraints in the base X10 language are conjunctions of equalities over immutable side-effect-free expressions. Compiler plugins may be installed to support other constraint languages and solvers.

The X10 compiler is implemented as an extension of Java using the Polyglot compiler framework [44]. Expressions used in constrained types are type-checked as normal non-dependent X10 expressions; no constraint solving is performed on these expressions. During type-checking, constraints are generated and solved using the built-in constraint solver or using solvers provided by plugins. The system allows types to constrained by conjunctions of constraints in different constraint languages. If constraints cannot be solved, an error is reported.

### 4.1 Constraint checking

After type-checking a constraint as a boolean expression `e`, the abstract syntax tree for the boolean expression is transformed into a conjunction of predicates, $\texttt{e}_1$ & … & $\texttt{e}_k$. Each conjunct $\texttt{e}_i$ is given to the installed constraint system plugins, which symbolically evaluate the expression to create an internal representation of the conjunct. If no constraint system can handle the conjunct, an error is reported.

To interoperate, the constraint solvers must share a common vocabulary: constraint terms `t` range over the properties of the base type, the final variables in scope at the type (including `this`), the special variable `self` representing a value of the type, and field selections `t.f`. All constraint systems are required to support the trivial constraint `true`, conjunction, existential quantification, and equality on constraint terms.

In this form, the constraint is represented as a conjunction of constraints from different theories. Constraints are checked for satisfiability using a Nelson–Oppen procedure [42]. After constructing a constraint-system specific representation of a conjunct, each plugin computes the set of term equalities entailed by the conjunct. These equalities are propagated to the other conjuncts, which are again checked for satisfiability and any new equalities generated are propagated. If a conjunct is found to be unsatisfiable, an error is reported.

During type-checking, the type checker needs to determine if the type `C{c}` is a subtype of `D{d}`. This is true if the base type `C` is a subtype of `D` and if the constraint `c` entails `d`. To check entailment, each constraint solver is asked if a given conjunct of `d` is entailed by `c`. If any report false, the entailment does not hold and the subtyping check fails.

### 4.2 Translation

After constraint-checking, the X10 code is translated to Java in a straightforward manner. Each dependent class is translated into a single class of the same name without dependent types. The explicit properties of the dependent class are translated into `public final` instance fields of the target class. A `property` statement in a constructor is translated to a sequence of assignments to initialize the property fields.

For each property, a getter method is also generated in the target Java class. Properties declared in interfaces are translated into getter method signatures. Subclasses implementing these interfaces thus provide the required properties by implementing the generated interfaces.

Usually, constrained types are simply translated to non-constrained types by erasure; constraints are checked statically and need no run-time representation. However, depen-

dent types may be used in casts and `instanceof` expressions. The language does not allow existential constraints to be used in run-time type tests; this allows the tests of constrained types to be implemented by evaluating the constraint with `self` bound to the expression being tested. For example, casts are translated as:

```
⟦e as C{c}⟧ =
  new Object() {
    C cast(C self) {
      if (⟦c⟧) return self;
      throw new ClassCastException(); }
  }.cast((C) ⟦e⟧)
```

Wrapping the evaluation of `c` in an anonymous class ensures the expression `e` is evaluated only once.

To support separate compilation, abstract syntax trees for constraints are embedded into the generated Java code, and from there into the generated class file. The compiler reconstructs dependent types in referenced class files from their ASTs.

# 5. Formal semantics

In this section we formalize a small fragment of X10, CFJ—an extension of Featherweight Java (FJ) [29] with constrained types—to illustrate the basic concepts behind constrained type-checking. A proof of soundness is given in the appendix.

The language is functional in that assignment is not admitted. However, it is not difficult to introduce the notion of mutable fields, and assignment to such fields. Since constrained types may only refer to immutable state, the validity of these types is not compromised by the introduction of state. Further, we do not formalize overloading of methods. Rather, as with FJ, we simply require that the input program be such that the class name `C` and method name `m` uniquely select the associated method on the class.

The language is defined over a constraint system $\mathcal{C}$ that includes equality constraints over final access paths, conjunction, existential quantification, and a vocabulary of formulas and predicates.

## 5.1 The Object constraint system

Given a program $P$, we now show how to build a larger constraint system $O(\mathcal{C})$ on top of $\mathcal{C}$ which captures constraints related to the object-oriented structure of $P$. $O$ includes the inference rules shown in Figure 3 for structual and subtyping constraints. In addition, $O(\mathcal{C})$ subsumes $\mathcal{C}$ in that if $\Gamma \vdash_{\mathcal{C}} \mathtt{c}$ then $\Gamma \vdash_{O} \mathtt{c}$.

The constraint `class(C)` is intended to be true for all classes `C` defined in the program. For a variable `x`, `fields(x)` is intended to specify the (complete) set of typed fields available to `x`. `x has I` is intended to specify that the member `I` (field or method) is available to `x`—for instance it is defined at the class at which `x` is declared or inherited by it, or it is available at the upper bound of a type variable. The judgment $\Gamma \vdash_{O} \mathtt{S <: T}$ is intended to hold if `S` is a subtype of `T` in the environment $\Gamma$.

We assume that the rules given are complete for defining the predicates `C<:D` and `C has I`, for classes `C`, `D` and members `I`; that is, if the rules cannot be used to establish $\vdash_O \mathtt{C <: D}$ ($\vdash_O \mathtt{C\ has\ I}$), then it is the case that $\vdash_O \mathtt{C} \not<: \mathtt{D}$ ($\vdash_O \neg(\mathtt{C\ has\ I})$). Such negative facts are important to establish *inconsistency* of assumptions (for instance, for the programming languages which permits the user to state constraints on type variables).

## 5.2 Judgments

In the following $\Gamma$ is a *well-typed context*, i.e., a finite, possibly empty sequence of formulas $\mathtt{x} : \mathtt{T}$ and constraints `c` satisfying:

1. for any formula $\phi$ in the sequence all variables `x` occurring in $\phi$ are defined by a declaration $\mathtt{x} : \mathtt{T}$ in the sequence to the left of $\phi$.
2. for any variable `x` , there is at most one formula $\mathtt{x} : \mathtt{T}$ in $\Gamma$.

The judgments of interest are as follows. (1) Type well-formedness: $\Gamma \vdash \mathtt{T\ type}$, (2) Subtyping: $\Gamma \vdash \mathtt{S <: T}$, (3) Typing: $\Gamma \vdash \mathtt{e} : \mathtt{T}$, (4) Method OK (method `M` is well-defined for the class `C`): $\Gamma \vdash \mathtt{M}$ OK in `C`, (5) Field OK (field $\mathtt{f} : \mathtt{T}$ is well-defined for the class `C`): $\Gamma \vdash \mathtt{f} : \mathtt{T}$ OK in `C` (6) Class OK: $\Gamma \vdash \mathtt{L}$ OK (class definition `L` is well-formed).

In defining these judgments we will use $\Gamma \vdash_O c$, the judgment corresponding to the Object constraint system. Recall that $O$ subsumes the underlying constraint system $\mathcal{C}$. For simplicity, we define $\Gamma \vdash \mathtt{c}$ to mean $\sigma(\Gamma) \vdash_O \mathtt{c}$, where the *constraint projection*, $\sigma(\Gamma)$ is defined as allows.

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

Above, in the third rule we assume that alpha-equivalence is used to choose the variable `x` from a set of variables that does not occur in the context $\Gamma$.

We say that a context $\Gamma$ is *consistent* if all (finite) subsets of $\{\sigma(\phi) \mid \Gamma \vdash \phi\}$ are consistent. In all type judgments presented below (T-CAST, T-FIELD etc) we make the implicit assumption that the context $\Gamma$ is consistent; if it is inconsistent, the rule cannot be used and the type of the given expression cannot be established (type-checking fails).

## 5.3 CFJ

The syntax and semantics of CFJ is presented in Figure 4. The syntax is essentially that of FJ with the following major exceptions. First, types may be constrained with a clause $\{\mathtt{c}\}$. Second both classes and methods may have constraint clauses `c`—in the case of classes, `c` is to be thought of as an invariant satisfied by all instances of the class, and in the case of methods, `c` is an additional condition that must be

**Structural constraints:**

$$\frac{\texttt{class C}(\ldots)\ \texttt{extends D}\{\ldots\} \in P}{\vdash_O \texttt{class(C)}}\ (\textsc{Class}) \qquad \vdash_O \texttt{new D}(\overline{\texttt{t}}).\texttt{f}_\texttt{i}==\texttt{t}_\texttt{i}\ (\textsc{Sel}) \qquad \frac{\Gamma \vdash_O \texttt{x} : \texttt{C}, \texttt{class(C)}}{\Gamma \vdash_O \mathit{inv}(\texttt{C},\texttt{x})}\ (\textsc{Inv}) \qquad \frac{\Gamma \vdash_O \texttt{fields(x)} = \overline{\texttt{f}} : \overline{\texttt{T}}}{\Gamma \vdash_O \texttt{x has f}_i : \texttt{T}_i}\ (\textsc{Field})$$

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

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

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

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

**Subtyping:**

$$\vdash_O \texttt{T} <: \texttt{T}\ (\textsc{S-Id}) \qquad \frac{\Gamma \vdash_O \texttt{T}_1 <: \texttt{T}_2, \texttt{T}_2 <: \texttt{T}_3}{\Gamma \vdash_O \texttt{T}_1 <: \texttt{T}_3}\ (\textsc{S-Trans}) \qquad \frac{\texttt{class C}(\ldots)\ \texttt{extends D}\{\ldots\} \in P}{\vdash_O \texttt{C} <: \texttt{D}}\ (\textsc{S-Extends}) \qquad \frac{\Gamma, \texttt{c} \vdash_O \texttt{S} <: \texttt{T}}{\Gamma \vdash_O \texttt{S}\{\texttt{c}\} <: \texttt{T}}\ (\textsc{S-Const-L})$$

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

For a class C and variable x, $\mathit{inv}(\texttt{C},\texttt{x})$ stands for the conjunction of class invariants for C and its supertypes, with this replaced by x.

**Figure 3.** The Object constraint system, $O$

satisfied by the receiver and the arguments of the method in order for the method to be invoked.

We assume a constraint system $\mathcal{C}$, with a vocabulary of predicates q and functions f. Constraints include true, conjunctions, existentials, predicates, and term equality. Constraints are well-formed if they are of the pre-given type o. The rules PRED and FUN ensure that predicates and formulas are well-formed and appeal to the constraint system $\mathcal{C}$.

The set of types includes classes C and is closed under constrained types (T{c}) and existential quantification (x:S;T). An object o is of type C (for C a class) if it is an instance of a subtype of C; it is of type T{c} if it is of type T and it satisfies the constraint c[o/self][4]; it is of type x:S;T if there is some object q of type S such that o is of type T[q/x] (treating at type as a syntactic expression).

The rules for well-formedness of types are straightforward, given the assumption that constraints are of type o.

***Typing rules.*** T-VAR is as expected, except that it asserts the constraint self==x which records the fact that any value of this type is known statically to be equal to x. This constraint is actually crucial—as we shall see in the other rules once we establish that an expression e is of a given type T, we "transfer" the type to a freshly chosen variable z. If in fact e has a static "name" x (i.e., e is known statically to be equal to x; that is, e is of type T{self==x}), then T-VAR lets us assert that z:T{self==x}, i.e., z equals x. Thus T-VAR provides an important base case for reasoning statically about equality of values in the environment.

We do away with the three casts provided in FJ in favor of a single cast, requiring only that e be of some type U. At run time e will be checked to see if it is actually of type T (see Rule R-CAST).

T-FIELD may be understood through "proxy" reasoning as follows. Given the context Γ assume the receiver e can be established to be of type S. Now we do not know the run-time value of e, so we shall assume that it is some fixed but unknown "proxy" value z (of type S) that is "fresh" in that it is not known to be related to any known value (i.e., those recorded in Γ). If we can establish that z has a field f of type T[5], then we can assert that e.f has type T and, further, that it equals z.f. Hence, we can assert that e.f has type (z:S; T{self==z.f}).

T-INVK has a very similar structure to T-FIELD: we use "proxy" reasoning for the receiver and the arguments of the method call. T-NEW also uses the same proxy reasoning; however, in this case we can establish that the resulting value is equal to new C($\overline{\texttt{v}}$) for some values $\overline{\texttt{v}}$ of the given type. The rule requires that the class invariant of C be established.

***Operational semantics.*** The operational semantics is essentially identical to FJ [29]. It is described in terms of a non-deterministic reduction relation on expressions. The only novelty is the use of the subtyping relation to check that the cast is satisfied. In CFJ, this test simply involves checking that the class of which the object is an instance is a

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

[5] Note from the definition of fields in $O$ (Figure 3) that all occurrences of this in the declared type of the field f will have been replaced by z.

**CFJ productions:**

$$\begin{array}{rl}
\text{(Class)} & \texttt{L} ::= \texttt{class C}(\overline{\texttt{f}}:\overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends N}\ \{\overline{\texttt{M}}\} \\
\text{(Method)} & \texttt{M} ::= \texttt{def m}(\overline{\texttt{x}}:\overline{\texttt{T}})\{\texttt{c}\}\texttt{:T = e;} \\
\text{(Exp.)} & \texttt{e} ::= \texttt{x} \mid \texttt{this} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid \texttt{e as T}
\end{array}$$

$$\begin{array}{rl}
\text{(Type)} & \texttt{S,T,U} ::= \texttt{N} \mid \texttt{T}\{\texttt{c}\} \mid \texttt{x:S;T} \\
\text{(N Type)} & \texttt{N} ::= \texttt{C} \mid \texttt{N}\{\texttt{c}\} \\
\text{(C Term)} & \texttt{t} ::= \texttt{x} \mid \texttt{self} \mid \texttt{this} \mid \texttt{t.f} \mid \texttt{new C}(\overline{\texttt{t}}) \mid \texttt{f}(\overline{\texttt{t}}) \\
\text{(Const.)} & \texttt{c,d} ::= \texttt{true} \mid \texttt{t==t} \mid \texttt{c,c} \mid \texttt{x:T; c} \mid \texttt{q}(\overline{\texttt{t}})
\end{array}$$

**Constraint well-formedness rules:**

$$\Gamma \vdash \texttt{true} : \texttt{o} \quad \text{(TRUE)} \qquad \frac{\Gamma \vdash \texttt{c}_0 : \texttt{o} \quad \Gamma \vdash \texttt{c}_1 : \texttt{o}}{\Gamma \vdash (\texttt{c}_0, \texttt{c}_1) : \texttt{o}} \quad \text{(AND)} \qquad \frac{\Gamma \vdash \texttt{t} : \texttt{T} \quad \Gamma \vdash \texttt{c}[\texttt{t}/\texttt{x}] : \texttt{o}}{\Gamma \vdash \texttt{x} : \texttt{T}; \texttt{c} : \texttt{o}} \quad \text{(EXISTS)}$$

$$\frac{\texttt{q}(\overline{\texttt{T}}) : \texttt{o} \in \mathcal{C} \quad \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{T}}}{\Gamma \vdash \texttt{q}(\overline{\texttt{T}}) : \texttt{o}} \quad \text{(PRED)} \qquad \frac{\texttt{f}(\overline{\texttt{T}}) : \texttt{T} \in \mathcal{C} \quad \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{T}}}{\Gamma \vdash \texttt{f}(\overline{\texttt{T}}) : \texttt{T}} \quad \text{(FUN)} \qquad \frac{\Gamma \vdash \texttt{t}_0 : \texttt{T}_0 \quad \Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \quad (\Gamma \vdash \texttt{T}_0 \texttt{<:} \texttt{T}_1 \vee \Gamma \vdash \texttt{T}_1 \texttt{<:} \texttt{T}_0)}{\Gamma \vdash \texttt{t}_0 \texttt{==} \texttt{t}_1 : o} \quad \text{(EQUALS)}$$

**Type well-formedness rules:**

$$\frac{\Gamma \vdash \texttt{class(C)}}{\Gamma \vdash \texttt{C type}} \quad \text{(CLASS)} \qquad \frac{\Gamma \vdash \texttt{S type}, \texttt{T type}}{\Gamma \vdash \texttt{x} : \texttt{S}; \texttt{T type}} \quad \text{(EXISTS-T)} \qquad \frac{\Gamma \vdash \texttt{T type} \quad \Gamma, \texttt{self} : \texttt{T} \vdash \texttt{c} : \texttt{o}}{\Gamma \vdash \texttt{T}\{\texttt{c}\}\ \texttt{type}} \quad \text{(DEP)}$$

**Typing rules:**

$$\Gamma, \texttt{x} : \texttt{T} \vdash \texttt{x} : \texttt{T}\{\texttt{self} == \texttt{x}\} \ \text{(T-VAR)} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{U} \quad \Gamma \vdash \texttt{T type}}{\Gamma \vdash \texttt{e as T} : \texttt{T}} \ \text{(T-CAST)} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{S} \quad \Gamma, \texttt{z} : \texttt{S} \vdash \texttt{z has f} : \texttt{T} \quad (\texttt{z fresh})}{\Gamma \vdash \texttt{e.f} : (\texttt{z} : \texttt{S}; \texttt{T}\{\texttt{self} == \texttt{z.f}\})} \ \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : \texttt{T}, \overline{\texttt{e}} : \overline{\texttt{T}} \\ \Gamma, \texttt{z} : \texttt{T}, \overline{\texttt{z}} : \overline{\texttt{T}} \vdash \texttt{z has m}(\overline{\texttt{z}} : \overline{\texttt{U}})\{\texttt{c}\} : \texttt{S} = \texttt{e}', \overline{\texttt{T}} \texttt{<:} \overline{\texttt{U}}, \texttt{c} \quad (\texttt{z},\overline{\texttt{z}}\ \text{fresh})\end{array}}{\Gamma \vdash \texttt{e.m}(\overline{\texttt{e}}) : (\texttt{z} : \texttt{T}; \overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{S})} \ \text{(T-INVK)} \qquad \frac{\begin{array}{c}\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}} \quad \vdash \texttt{class(C)} \\ \Gamma, \texttt{z} : \texttt{C} \vdash \texttt{fields}(\texttt{z}) = \overline{\texttt{f}} : \overline{\texttt{U}} \quad (\texttt{z},\overline{\texttt{z}}\ \text{fresh}) \\ \Gamma, \texttt{z} : \texttt{C}, \overline{\texttt{z}} : \overline{\texttt{T}}, \texttt{z.}\overline{\texttt{f}} = \overline{\texttt{z}} \vdash \overline{\texttt{T}} \texttt{<:} \overline{\texttt{U}}, \mathit{inv}(\texttt{C}, \texttt{z})\end{array}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) : \texttt{C}\{\overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{new C}(\overline{\texttt{z}}) = \texttt{self}, \mathit{inv}(\texttt{C}, \texttt{self})\}} \ \text{(T-NEW)}$$

$$\frac{\texttt{this} : \texttt{C} \vdash \texttt{c} : \texttt{o} \quad \texttt{this} : \texttt{C}, \overline{\texttt{x}} : \overline{\texttt{U}}, \texttt{c} \vdash \texttt{T type}, \overline{\texttt{U}}\ \texttt{type}, \texttt{e} : \texttt{S}, \texttt{S} \texttt{<:} \texttt{T}}{\texttt{def m}(\overline{\texttt{x}} : \overline{\texttt{U}})\{\texttt{c}\} : \texttt{T} = \texttt{e};\ \text{OK in } C} \ \text{(METHOD OK)} \qquad \frac{\overline{\texttt{M}}\ \text{OK in C} \quad \texttt{this} : \texttt{C} \vdash \texttt{c} : \texttt{o} \quad \texttt{this} : \texttt{C}, \texttt{c} \vdash \overline{\texttt{T}}\ \texttt{type}, \texttt{N type}}{\texttt{class C}(\overline{\texttt{f}} : \overline{\texttt{T}})\{\texttt{c}\}\ \texttt{extends N}\{\overline{\texttt{M}}\}\ \text{OK}} \ \text{(CLASS OK)}$$

**Transition rules:**

$$\frac{\texttt{x} : \texttt{C} \vdash \texttt{fields}(\texttt{x}) = \overline{\texttt{f}} : \overline{\texttt{T}}}{(\texttt{new C}(\overline{\texttt{e}}))\texttt{.f}_\texttt{i} \to \texttt{e}_\texttt{i}} \ \text{(R-FIELD)} \qquad \frac{\texttt{x} : \texttt{C} \vdash \texttt{x has m}(\overline{\texttt{x}} : \overline{\texttt{T}})\{\texttt{c}\} : \texttt{T} = \texttt{e}}{(\texttt{new C}(\overline{\texttt{e}}))\texttt{.m}(\overline{\texttt{d}}) \to \texttt{e}[\texttt{new C}(\overline{\texttt{e}}), \overline{\texttt{d}}/\texttt{this}, \overline{\texttt{x}}]} \ \text{(R-INVK)}$$

$$\frac{\texttt{e} \to \texttt{e}'}{\texttt{e.f}_i \to \texttt{e}'\texttt{.f}_i} \ \text{(RC-FIELD)} \qquad \frac{\texttt{e} \to \texttt{e}'}{\texttt{e.m}(\overline{\texttt{e}}) \to \texttt{e}'\texttt{.m}(\overline{\texttt{e}})} \ \text{(RC-INVK-RECV)}$$

$$\frac{\vdash \texttt{C}\{\texttt{self} == \texttt{new C}(\overline{\texttt{d}})\} \texttt{<:} \texttt{T}}{\texttt{new C}(\overline{\texttt{d}})\ \texttt{as T} \to \texttt{new } C(\overline{\texttt{d}})} \ \text{(R-CAST)} \qquad \frac{\texttt{e}_i \to \texttt{e}_i{}'}{\texttt{e.m}(\ldots, \texttt{e}_i, \ldots) \to \texttt{e.m}(\ldots, \texttt{e}_i', \ldots)} \ \text{(RC-INVK-ARG)}$$

$$\frac{\texttt{e} \to \texttt{e}'}{\texttt{e as T} \to \texttt{e}'\ \texttt{as T}} \ \text{(RC-CAST)} \qquad \frac{\texttt{e}_i \to \texttt{e}_i{}'}{\texttt{new C}(\ldots, \texttt{e}_i, \ldots) \to \texttt{new C}(\ldots, \texttt{e}_i', \ldots)} \ \text{(RC-NEW-ARG)}$$

**Figure 4.** Semantics of CFJ

subclass of the class specified in the given type; in richer languages with richer notions of type this operation may involve run-time constraint solving using the fields of the object.

### 5.4 Results

The following results hold for CFJ.

THEOREM 5.1 (Subject Reduction). *If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \to \mathtt{e}'$ *then for some type* $\mathtt{S}$, $\Gamma \vdash \mathtt{e}' : \mathtt{S}, S <: T$.

The theorem needs the Substitution Lemma:

LEMMA 5.2. *The following is a derived rule:*

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

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

THEOREM 5.3 (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 cast sub-expression which is stuck,*
3. *there exists an* $\mathtt{e}'$ *s.t.* $\mathtt{e} \to \mathtt{e}'$.

THEOREM 5.4 (Type soundness). *If* $\vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e}$ *reduces to a normal form* $\mathtt{e}'$ *then either* $\mathtt{e}'$ *is a value* $\mathtt{v}$ *and* $\vdash \mathtt{v} : \mathtt{S}, \mathtt{S} <: \mathtt{T}$ *or* $\mathtt{e}'$ *contains a stuck cast sub-expression.*

## 6. Related work

***Constraint-based type systems.*** The use of constraints in type systems has a history going back to Mitchell [40] and Reynolds [54]. These and subsequent systems are based on constraints over types, but not over values. Constraint-based type systems for ML-like languages [60, 52] lead to HM($X$) [58], 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. 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.

Constrained types permit *user-defined* predicates and functions, allowing the user to enrich the constraint system, and hence the power of the compile-time type-checker, with application-specific constraints using a constraint programming language such as CLP($\mathcal{C}$) [30] or RCC($\mathcal{C}$) [31].

***Dependent types.*** Dependent type systems [62, 16, 38, 6] parametrize types on values. Constrained types are a form of refinement type [24, 1, 32, 28, 19, 20, 55]. Introduced by Freeman and Pfenning [24], refinement types are dependent types that extend a base type system through constraints on values.

Our work is closely related to DML, [62], an extension of ML with dependent types. DML is also built parametrically on a constraint solver. Types are refinement types; they do not affect the operational semantics and erasing the constraints yields a legal DML program. The most obvious distinction between DML and constrained types lies in the target domain: DML is designed for functional programming whereas constrained types are designed for imperative, concurrent object-oriented languages. But there are several other crucial differences as well.

DML achieves its separation between compile-time and run-time processing by not permitting program variables to be used in types. Instead, a parallel set of (universally or existentially quantified) "index" variables are introduced. Second, DML permits only variables of basic index sorts known to the constraint solver (e.g., `bool`, `int`, `nat`) to occur in types. In contrast, constrained types permit program variables at any type to occur in constrained types. As with DML only operations specified by the constraint system are permitted in types. However, these operations always include field selection and equality on object references. Note that DML-style constraints are easily encoded in constrained types.

Logically qualified types, or liquid types [55], 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: two liquid types are in the subtyping relation if their base types are and if one type's qualifier implies the other's. The Hindley–Milner type inference algorithm is used to infer base types; these types are used as templates for inference of the liquid types. The types of certain expressions are over-approximated to ensure inference is decidable. To improve precision of the inference algorithm, and hence to reduce the annotation burden on the programmer, the type system is path sensitive. X10 does not (yet) support type inference.

Hybrid type-checking [19, 20] introduced another refinement type system. While typing is undecidable, dynamic checks are inserted into the program when necessary if the type-checker (which includes a constraint solver) cannot determine type safety statically. In X10, dynamic type checks, including tests of dependent constraints, are inserted only at explicit casts or `instanceof` expressions; constraint solving is performed at compile time.

Theorem provers have also been integrated into the programming language. For instance, Concoqtion [22] extends types in OCaml [34] with constraints written as Coq [15] rules. Constraints thus have a different syntax, representation, and behavior than the rest of the language. Proofs must be provided to satisfy the type checker. In contrast, X10 supports a more limited constraint language that can be checked by a constraint solver during compilation.

ESC/Java [21] allow programmers to write object invariants and pre- and post-conditions that are enforced statically by the compiler using an automated theorem prover. Static checking is undecidable and, in the presence of loops, is unsound (but still useful) unless the programmer supplies loop

invariants. Unlike X10, ESC/Java can enforce invariants on mutable state.

Constraints in X10 are over final access paths. Several other languages have dependent types defined over final access paths [17, 47, 50, 45, 49, 48, 14, 25]. In many of these languages, dependent path types are used to enforce type soundness for virtual classes [35, 36, 18] or similar mechanisms. Jif [41, 13] uses dependent types over final access paths to enforce security properties: the security policy of an expression may depend on the policies of other variables in the program. Aspects of these type systems can be encoded using equality constraints in X10. For example, for a final access path `p`, `p.type` in Scala is the singleton type containing the object `p`. Scala's `p.type` can be encoded in X10 using an equality constraint `C{self == p}`, where `C` is a supertype of `p`'s static type.

***Pluggable types.*** In X10, constraint system plugins can provide a constraint solver to check consistency and entailment of the extended constraint language.

Pluggable and optional type systems were proposed by Bracha [9] and provide a means of extending the base language's type system. In Bracha's proposal, type annotations, implemented in compiler plugins, may only reject programs statically that might otherwise have dynamic type errors; they may not change the run-time semantics of the language. Java annotations [26, 33] may be used to extend the Java type system with compiler plugins.

Other approaches, such as user-defined type qualifiers [23, 12] or JavaCOP [4] allow programmers to declaratively specify new typing rules in a meta language rather than through plugins.

## 7. Conclusion and future work

We have presented the design and implementation of constrained types in X10. The design considerably enriches the space of statically checkable types expressible in the language. This is particularly important for data-structures such as lists and arrays. Several examples of constrained types were presented. Constrained types have been implemented in X10 and used for place types, clocked types, and array types.

The implementation supports extension with constraint solver plugins. In future work, we plan to further investigate optimizations enabled by constrained types. We also plan to explore type inference for constrained types and to pursue more expressive constraint systems and extensions of constrained types for handling mutable state, control flow, and effects.

## Acknowledgments

The authors thank Radha Jagadeesan, Norman Cohen, Pradeep Varma, Satish Chandra, Martin Hirzel, Igor Peshansky, Lex Spoon, Vincent Cave, Vivek Sarkar, and the X10 team for fruitful discussions and implementation of the X10 compiler and examples. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

## References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 163–173, January 1994.

[2] Hassan Ait-Kaci. *A lattice theoretic approach to computation based on a calculus of partially ordered type structures (property inheritance, semantic nets, graph unification)*. PhD thesis, University of Pennsylvania, 1984.

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

[4] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 2006 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2006.

[5] David Aspinall and Martin Hofmann. *Dependent Types*, chapter 2. In Pierce [51], 2004.

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

[7] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* $16^{th}$ *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[8] Clark Barrett, Cesare Tinelli, Alexander Fuchs, Yeting Ge, George Hagen, and Dejan Jovanovic. CVC3. `http://www.cs.nyu.edu/acsys/cvc3`.

[9] Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[10] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[11] Craig Chambers. Predicate classes. In *ECOOP '93 Conference Proceedings*, 1993.

[12] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–95, 2005.

[13] Stephen Chong, Andrew C. Myers, K. Vikram, and Lantian Zheng. Jif reference manual, Jif 3.0.0 version.

http://www.cs.cornell.edu/jif, June 2006.

[14] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 121–134, New York, NY, USA, 2007. ACM Press.

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

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

[17] Erik Ernst. *gbeta: A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[18] 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.

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

[20] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 2006.

[21] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[22] Seth Fogarty, Emir Pašalić, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 112–121, January 2007.

[23] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 1–12. ACM Press, June 2002.

[24] 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.

[25] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 133–152. ACM, 2007.

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

[27] Christian Grothoff, Jens Palsberg, and Vijay Saraswat. Safe arrays via regions and dependent types. Technical Report RC23911, IBM T.J. Watson Research Center, 2006.

[28] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 410–423, 1996.

[29] 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.

[30] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119. ACM Press, New York (NY), USA, 1987.

[31] Radha Jagadeesan, Gopalan Nadathur, and Vijay A. Saraswat. Testing concurrent systems: An interpretation of intuitionistic logic. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS*, volume 3821 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 2005.

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

[33] JSR 308: Annotations on Java types. http://jcp.org/en/jsr/detail?id=308.

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

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

[36] 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.

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

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

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

[40] 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.

[41] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[42] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2), October 1979.

[43] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

[44] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th*

*International Conference, CC 2003*, number 2622 in LNCS, pages 138–152. Springer-Verlag, April 2003.

[45] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software extension. In *Proceedings of the 2006 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pages 21–36, Portland, OR, October 2006.

[46] Nathaniel Nystrom and Vijay Saraswat. An annotation and compiler plugin system for X10. Technical Report RC24198, IBM T.J. Watson Research Center, 2007.

[47] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language, June 2004. `http://scala.epfl.ch/docu/files/ScalaOverview.pdf`.

[48] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 201–224. Springer-Verlag, July 2003.

[49] Martin Odersky and Christoph Zenger. Nested types. In *8th Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

[50] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. OOPSLA '05*, pages 41–57, San Diego, CA, USA, October 2005.

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

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

[53] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, pages 866–869. Cambridge University Press, 1992. Successive overrelaxation (SOR).

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

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

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

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

[58] 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.

[59] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM/SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, 1997.

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

[61] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 249–257, Montreal, Canada, June 1998.

[62] 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.

## A. Soundness

Here we prove a soundness theorem for CFJ.

LEMMA A.1 (Substitution Lemma). *The following is a derived rule:*

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

PROOF. Straightforward. □

LEMMA A.2 (Weakening). *If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$*, then* $\Gamma, \mathtt{x} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$.

PROOF. Straightforward. □

LEMMA A.3. *If* $\Gamma \vdash \mathtt{S} <: \mathtt{T}$*, and* $\Gamma, \mathtt{z} : \mathtt{S} \vdash \mathtt{fields}(\mathtt{z}) = \overline{\mathtt{F}}_1$*, and* $\Gamma, \mathtt{z} : \mathtt{T} \vdash \mathtt{fields}(\mathtt{z}) = \overline{\mathtt{F}}_2$*, then* $\overline{\mathtt{F}}_2$ *is a prefix of* $\overline{\mathtt{F}}_1$.

PROOF. Immediate from the rules of *O*. □

LEMMA A.4. *If* $\Gamma \vdash \mathtt{S} <: \mathtt{T}$*, and* $\Gamma, \mathtt{z} : \mathtt{T} \vdash \mathtt{z}\ \mathtt{has}\ \mathtt{I}$*, then* $\Gamma, \mathtt{z} : \mathtt{T} \vdash \mathtt{z}\ \mathtt{has}\ \mathtt{I}$.

PROOF. Immediate from the rules of *O*. □

LEMMA A.5. *If* $\Gamma \vdash \mathtt{S} <: \mathtt{T}$*, then*

$$(\mathtt{z} : \mathtt{S};\ \mathtt{c}_0)[\mathtt{x}/\mathtt{self}] \vdash_{\mathcal{C}} (\mathtt{z} : \mathtt{T};\ \mathtt{c}_0)[\mathtt{x}/\mathtt{self}]$$

*where* x *is fresh.*

PROOF. Straightforward. □

LEMMA A.6. *If* $\Gamma \vdash \mathtt{S} <: \mathtt{T}$*, and* $\sigma(\Gamma, \mathtt{f} : \mathtt{T}) \vdash_{\mathcal{C}} \mathtt{c}_0$*, then* $\sigma(\Gamma, \mathtt{f} : \mathtt{S}) \vdash_{\mathcal{C}} \mathtt{c}_0$*,*

PROOF. From $\Gamma \vdash \mathtt{S} <: \mathtt{T}$, we must have $\mathtt{S} = \mathtt{C}\{\mathtt{c}\}$ and $\mathtt{T} = \mathtt{D}\{\mathtt{d}\}$ where $\mathtt{C} <: \mathtt{D}$ and $\sigma(\Gamma, \mathtt{x} : \mathtt{C}\{\mathtt{c}\}) \vdash_{\mathcal{C}} \mathtt{d}[\mathtt{x}/\mathtt{self}]$ with x fresh. From the definition of $\sigma(\cdot)$ we have

$$\sigma(\Gamma, \mathtt{f} : \mathtt{S}) = \sigma(\Gamma), c[\mathtt{f}/\mathtt{self}], \mathit{inv}(C, \mathtt{f})$$

and

$$\sigma(\Gamma, \mathtt{f} : \mathtt{T}) = \sigma(\Gamma), d[\mathtt{f}/\mathtt{self}], \mathit{inv}(D, \mathtt{f}).$$

From $\Gamma \vdash S <: T$ we have $\sigma(\Gamma, x : C\{c\}) \vdash_{\mathcal{C}} d[f/\texttt{self}]$. Additionally, from the definition of $inv(C,f)$ and from $C <: D$, we have that $inv(C,f)$ is a constraint that has $inv(D,f)$ as a conjunct so $inv(C,f) \vdash_{\mathcal{C}} inv(D,f)$. We conclude $\sigma(\Gamma, \texttt{f} : \texttt{S}) \vdash_{\mathcal{C}} \sigma(\Gamma, \texttt{f} : \texttt{T})$. We have that $\vdash_{\mathcal{C}}$ is transitive so from $\sigma(\Gamma, \texttt{f} : \texttt{S}) \vdash_{\mathcal{C}} \sigma(\Gamma, \texttt{f} : \texttt{T})$ and $\sigma(\Gamma, \texttt{f} : \texttt{T}) \vdash_{\mathcal{C}} c_0$, we have $\sigma(\Gamma, \texttt{f} : \texttt{S}) \vdash_{\mathcal{C}} c_0$. □

LEMMA A.7. *if* $\Gamma, \texttt{f} : \texttt{T} \vdash U <: U'$, *and* $\Gamma \vdash S <: T$, *then* $\Gamma, \texttt{f} : \texttt{S} \vdash U <: U'$.

PROOF. From $\Gamma, \texttt{f} : \texttt{T} \vdash U <: U'$ we must have $U = C\{c\}$ and $U' = D\{d\}$ where $C <: D$ and

$$\sigma(\Gamma, \texttt{f} : \texttt{T}, x : C\{c\}) \vdash_{\mathcal{C}} d[x/\texttt{self}]$$

with $x$ fresh. From Lemma A.6, $\Gamma \vdash S <: T$, and

$$\sigma(\Gamma, \texttt{f} : \texttt{T}, x : C\{c\}) \vdash_{\mathcal{C}} d[x/\texttt{self}],$$

we have

$$\sigma(\Gamma, \texttt{f} : \texttt{S}, x : C\{c\}) \vdash_{\mathcal{C}} d[x/\texttt{self}].$$

So we can use $C <: D$ and

$$\sigma(\Gamma, \texttt{f} : \texttt{S}, x : C\{c\}) \vdash_{\mathcal{C}} d[x/\texttt{self}]$$

to derive $\Gamma, \texttt{f} : \texttt{S} \vdash U <: U'$. □

LEMMA A.8. *if* $\Gamma \vdash S <: T$, *then* $\Gamma \vdash E\{z : S; c_0\} <: E\{z : T; c_0\}$.

PROOF. To prove the desired conclusion $E\{z : S; c_0\} <: E\{z : T; c_0\}$ we need to show that

$$\sigma(\Gamma, x : E\{z : S; c_0)\}) \vdash_{\mathcal{C}} (z : T; c_0)[x/\texttt{self}].$$

We have

$$\sigma(\Gamma, x : E\{z : S; c_0\}) = \sigma(\Gamma), (z : S; c_0)[x/\texttt{self}], inv(E, x).$$

From Lemma A.5 and $\Gamma \vdash S <: T$, we have

$$(z : S; c_0)[x/\texttt{self}] \vdash_{\mathcal{C}} (z : T; c_0)[x/\texttt{self}].$$

From

$$\sigma(\Gamma, x : E\{z : S; c_0\}) = \sigma(\Gamma), (z : S; c_0)[x/\texttt{self}], inv(E, x)$$

and

$$(z : S; c_0)[x/\texttt{self}] \vdash_{\mathcal{C}} (z : T; c_0)[x/\texttt{self}],$$

we conclude

$$\sigma(\Gamma, x : E\{z : S; c_0\}) \vdash_{\mathcal{C}} (z : T; c_0)[x/\texttt{self}].$$

□

LEMMA A.9. $\Gamma \vdash (\texttt{x} : \texttt{S}; \texttt{T\{c\}}) \equiv \texttt{T\{x: S; c\}}$.

THEOREM A.10 (Subject Reduction). *If* $\Gamma \vdash \texttt{e} : \texttt{V}$ *and* $\texttt{e} \to \texttt{e}'$, *then for some type* $\texttt{V}'$, $\Gamma \vdash \texttt{e}' : \texttt{V}'$ *and* $\Gamma \vdash \texttt{V}' <: \texttt{V}$.

PROOF. We proceed by induction on the structure of the derivation of $\Gamma \vdash \texttt{e} : \texttt{T}$. We now have five cases depending on the last rule used in the derivation of $\Gamma \vdash \texttt{e} : \texttt{T}$.

- T-VAR: The expression cannot take a step, so the conclusion is immediate.
- T-CAST: We have two subcases.
  - R-CAST: For the expression $\texttt{o as V}$, where $\texttt{o} = \texttt{new C}(\overline{\texttt{d}})$, we have from T-NEW that

    $$\Gamma \vdash \texttt{o} : \texttt{C}\{\overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{new C}(\overline{\texttt{z}}) = \texttt{self}; inv(\texttt{C}, \texttt{self})\}.$$

    Additionally, we have from R-CAST that

    $$\vdash \texttt{C}\{\texttt{new C}(\overline{\texttt{d}}) = \texttt{self}\} <: \texttt{V}.$$

    We now choose

    $$\texttt{V}' = \texttt{C}\{\overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{new C}(\overline{\texttt{z}}) = \texttt{self}; inv(\texttt{C}, \texttt{self})\}.$$

    From S-EXISTS-L,

    $$\Gamma \vdash \overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{C}\{\texttt{new C}(\overline{\texttt{z}}) = \texttt{self}; inv(\texttt{C}, \texttt{self})\} <: \texttt{C}\{\texttt{new C}(\overline{\texttt{d}}) = \texttt{self}\}.$$

    Lemma A.9,

    $$\Gamma \vdash \texttt{C}\{\overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{new C}(\overline{\texttt{z}}) = \texttt{self}; inv(\texttt{C}, \texttt{self})\} \equiv \overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{C}\{\texttt{new C}(\overline{\texttt{z}}) = \texttt{self}; inv(\texttt{C}, \texttt{self})\}$$

    From S-TRANS, $\Gamma \vdash \texttt{V}' <: \texttt{V}$.
  - RC-CAST: For the expression $\texttt{o as V}$, we have from T-CAST that $\Gamma \vdash \texttt{o} : \texttt{U}$. Additionally, we have from RC-CAST that $\texttt{o} \to \texttt{o}'$. From the induction hypothesis, we have $\texttt{U}'$ such that $\Gamma \vdash \texttt{o}' : \texttt{U}'$ and $\Gamma \vdash \texttt{U}' <: \texttt{U}$. We now choose $\texttt{V}' = \texttt{V}$. From $\Gamma \vdash \texttt{o}' : \texttt{U}'$ and T-CAST we derive $\Gamma \vdash \texttt{o}' \texttt{ as V} : \texttt{V}$. From $\texttt{V}' = \texttt{V}$ and S-ID we have $\Gamma \vdash \texttt{V}' <: \texttt{V}$.
- T-NEW: We have a single case.
  - RC-NEW-ARG: For the expression $\texttt{new C}(\overline{\texttt{e}})$, we have from T-NEW that $\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}}$, $\vdash \texttt{class}(C)$, $\Gamma \vdash \texttt{z} : \texttt{C} \vdash \texttt{fields}(\texttt{z}) = \overline{\texttt{f}} : \overline{\texttt{S}}$, $\Gamma \vdash \texttt{z} : \texttt{C}, \overline{\texttt{z}} : \overline{\texttt{T}}, \texttt{z}.\overline{\texttt{f}} = \overline{\texttt{z}} \vdash \overline{\texttt{T}} <: \overline{\texttt{S}}, inv(\texttt{C}, \texttt{z})$.
    We choose $\texttt{V} = \texttt{C}\{\overline{\texttt{z}} : \overline{\texttt{T}}; \texttt{new C}(\overline{\texttt{z}}) = \texttt{self}, inv(\texttt{C}, \texttt{self})\}$.
    Additionally, we have from RC-NEW-ARG that $\texttt{e}_i \to \texttt{e}'_i$. From the induction hypothesis, we have $\texttt{U}_i$ such that $\Gamma \vdash \texttt{e}'_i : \texttt{U}_i$ and $\Gamma \vdash U_i <: T_i$.
    For all $j$ except $i$, define $\texttt{U}_j = \texttt{T}_j$ and $\texttt{e}'_j = \texttt{e}_j$. We have $\Gamma \vdash \overline{\texttt{e}}' : \overline{\texttt{U}}$ and $\Gamma \vdash \overline{\texttt{U}} <: \overline{\texttt{T}}$.
    From Lemma A.2, we have $\Gamma \vdash \texttt{z} : \texttt{C}, \overline{\texttt{z}} : \overline{\texttt{U}}, \texttt{z}.\overline{\texttt{f}} = \overline{\texttt{z}} \vdash \overline{\texttt{U}} <: \overline{\texttt{T}}$.
    From T-NEW, we have $\Gamma \vdash \texttt{z} : \texttt{C}, \overline{\texttt{z}} : \overline{\texttt{T}}, \texttt{z}.\overline{\texttt{f}} = \overline{\texttt{z}} \vdash \overline{\texttt{T}} <: \overline{\texttt{S}}$. From Lemma A.7, $\Gamma \vdash \texttt{z} : \texttt{C}, \overline{\texttt{z}} : \overline{\texttt{U}}, \texttt{z}.\overline{\texttt{f}} = \overline{\texttt{z}} \vdash \overline{\texttt{T}} <: \overline{\texttt{S}}$.

From S-TRANS, we have $\Gamma \vdash \mathtt{z} : \mathtt{C}, \overline{\mathtt{z}} : \overline{\mathtt{U}}, \mathtt{z}.\overline{\mathtt{f}} = \overline{\mathtt{z}} \vdash \overline{\mathtt{U}} <: \overline{\mathtt{S}}$.
From T-NEW, $\Gamma \vdash \mathtt{z} : \mathtt{C}, \overline{\mathtt{z}} : \overline{\mathtt{T}}, \mathtt{z}.\overline{\mathtt{f}} = \overline{\mathtt{z}} \vdash \mathit{inv}(\mathtt{C}, \mathtt{z})$.
From Lemma A.7, $\Gamma \vdash \mathtt{z} : \mathtt{C}, \overline{\mathtt{z}} : \overline{\mathtt{U}}, \mathtt{z}.\overline{\mathtt{f}} = \overline{\mathtt{z}} \vdash \mathit{inv}(\mathtt{C}, \mathtt{z})$.
Thus, by T-NEW, $\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}') : \mathtt{C}\{\overline{\mathtt{z}} : \overline{\mathtt{U}}; \mathtt{new}\ \mathtt{C}(\overline{\mathtt{z}}) = \mathtt{self}, \mathit{inv}(\mathtt{C}, \mathtt{self})\}$ and we choose $\mathtt{V}' = \mathtt{C}\{\overline{\mathtt{z}} : \overline{\mathtt{U}}; \mathtt{new}\ \mathtt{C}(\overline{\mathtt{z}}) = \mathtt{self}, \mathit{inv}(\mathtt{C}, \mathtt{self})\}$.
From Lemma A.5, we have $\Gamma \vdash \mathtt{V}' <: \mathtt{V}$.

- T-FIELD: We have two subcases.
  - R-FIELD: For the expression $(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{f_i}$, we have from T-FIELD that $\Gamma \vdash \mathtt{e} : \mathtt{S}$ and $\Gamma, \mathtt{z} : \mathtt{S} \vdash \mathtt{z}\ \mathtt{has}\ \mathtt{f}_i : \mathtt{U}_i$.
    Let $\mathtt{V} = (\mathtt{z} : \mathtt{S}; \mathtt{U}_i\{\mathtt{self} == \mathtt{z}.\mathtt{f}_i\})$. $\mathtt{z}$ is fresh.
    We have $\mathtt{S} = \mathtt{C}\{\overline{\mathtt{z}} : \overline{\mathtt{T}}; \mathtt{new}\ \mathtt{C}(\overline{\mathtt{z}}) = \mathtt{self}, \mathit{inv}(\mathtt{C}, \mathtt{self})\}$.
    From T-NEW, we have $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ and $\Gamma \vdash \mathtt{z} : \mathtt{C}, \overline{\mathtt{z}} : \overline{\mathtt{T}}, \mathtt{z}.\overline{\mathtt{f}} = \overline{\mathtt{z}} \vdash \mathtt{T}_i <: \mathtt{U}_i$.
    From $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$, we have $\Gamma \vdash \mathtt{e}_i : \mathtt{T}_i$. We now choose $\mathtt{V}' = \mathtt{T}_i$.
    By T-NEW, $\Gamma, \mathtt{z} : \mathtt{C}, \overline{\mathtt{z}} : \overline{\mathtt{T}}, \mathtt{z}.\overline{\mathtt{f}} = \overline{\mathtt{z}} \vdash \mathtt{T}_i <: \mathtt{U}_i$.
    By S-CONST-R, $\Gamma, \mathtt{z} : \mathtt{C}, \overline{\mathtt{z}} : \overline{\mathtt{T}}, \mathtt{z}.\overline{\mathtt{f}} = \overline{\mathtt{z}} \vdash \mathtt{T}_i <: \mathtt{U}_i\{\mathtt{self} = \mathtt{z}_i\}$.
    Since $\mathtt{z}.\mathtt{f}_i = \mathtt{z}_i$, by application of S-CONST-R, S-CONST-L, and S-ID, we have $\Gamma, \mathtt{z} : \mathtt{C}, \overline{\mathtt{z}} : \overline{\mathtt{T}}, \mathtt{z}.\overline{\mathtt{f}} = \overline{\mathtt{z}} \vdash \mathtt{T}_i <: \mathtt{U}_i\{\mathtt{self} = \mathtt{z}.\mathtt{f}_i\}$.
    We can then show via S-EXISTS-R, $\Gamma \vdash \mathtt{T}_i <: (\mathtt{z} : \mathtt{S}; \mathtt{U}_i\{\mathtt{self} = \mathtt{z}.\mathtt{f}_i\})$, or more simply $\Gamma \vdash \mathtt{V}' <: \mathtt{V}$.
  - RC-FIELD: Follows from the induction hypothesis and application of Lemma A.8.
- T-INVK: We have three subcases.
  - R-INVK: For simplicity, define $\mathtt{d}_0 = \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$. For the expression $\mathtt{d}_0.\mathtt{m}(\overline{\mathtt{d}})$ we have from T-INVK that

$$\begin{array}{l}\Gamma \vdash \mathtt{d}_0 : \mathtt{T}_0 \\ \Gamma \vdash \mathtt{d}_{1:n} : \mathtt{T}_{1:n} \\ \Gamma, \mathtt{z}_{0:n} : \mathtt{T}_{0:n} \vdash \mathtt{z}_0\ \mathtt{has}\ \mathtt{m}(\mathtt{z}_{1:n} : \mathtt{U}_{1:n})\{c\} : \mathtt{S} = \mathtt{e} \\ \Gamma, \mathtt{z}_{0:n} : \mathtt{T}_{0:n} \vdash \mathtt{T}_{1:n} <: \mathtt{U}_{1:n} \\ \Gamma, \mathtt{z}_{0:n} : \mathtt{T}_{0:n} \vdash \mathtt{c}\end{array}$$

    where $\mathtt{z}_{0:n}$ is fresh. By T-NEW, we have $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{A}}$ and $\mathtt{T}_0 = \mathtt{C}\{\overline{\mathtt{z}} : \overline{\mathtt{A}}; \mathtt{self} = \mathtt{new}\ \mathtt{C}(\overline{\mathtt{z}}), \mathit{inv}(\mathtt{C}, \mathtt{self})\}$.
    Since $\Gamma, \mathtt{z}_{0:n} : \mathtt{T}_{0:n} \vdash \mathtt{z}_0\ \mathtt{has}\ \mathtt{m}(\mathtt{z}_{1:n} : \mathtt{U}_{1:n})\{c\} : \mathtt{S} = \mathtt{e}$, and $\Gamma, \mathtt{z}_{0:n} : \mathtt{T}_{0:n} \vdash \mathtt{T}_{1:n} <: \mathtt{U}_{1:n}$, by Lemma **??**, we have $\Gamma \vdash \mathtt{e} : (\mathtt{z}_{0:n} : \mathtt{T}_{0:n};\ \mathtt{S})$.
    Choose $\mathtt{V} = (\mathtt{z}_{0:n} : \mathtt{T}_{0:n};\ \mathtt{S})$.
    From R-INVK, we have $\mathtt{d}_0.\mathtt{m}(\overline{\mathtt{d}}) \to \mathtt{e}[\mathtt{d}_0, \overline{\mathtt{d}}/\mathtt{this}, \overline{\mathtt{z}}]$.
    By Lemma A.1, $\Gamma \vdash \mathtt{e}[\mathtt{d}_0, \overline{\mathtt{d}}/\mathtt{this}, \overline{\mathtt{z}}] : \mathtt{V}'$, and $\Gamma \vdash \mathtt{V}' <: \mathtt{z}_{0:n} : \mathtt{T}_{0:n};\ \mathtt{S}$.
  - RC-INVK-RECV: Follows from the induction hypothesis and application of Lemma A.8.
  - RC-INVK-ARG: Follows from the induction hypothesis and application of Lemma A.8.

□

Let the normal form of expressions be given by *values* $\mathtt{v} ::= \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$.

THEOREM A.11 (Progress). *If* $\vdash \mathtt{e} : \mathtt{T}$*, then one of the following conditions holds:*

1. $\mathtt{e}$ *is a value* $\mathtt{v}$*,*
2. $\mathtt{e}$ *contains a subexpression* $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})\ \mathtt{as}\ \mathtt{T}$ *such that* $\nvdash \mathtt{C} <: \mathtt{T}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})/\mathtt{self}]$*,*
3. *there exists* $\mathtt{e}'$ *s.t.* $\mathtt{e} \to \mathtt{e}'$*.*

PROOF. The proof has a structure that is similar to the proof of Subject Reduction; we omit the details. □

THEOREM A.12 (Type Soundness). *If* $\vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \to^* \mathtt{e}'$*, with* $\mathtt{e}'$ *in normal form, then* $\mathtt{e}'$ *is either (1) a value* $\mathtt{v}$ *with* $\vdash \mathtt{v} : \mathtt{S}$ *and* $\vdash \mathtt{S} <: \mathtt{T}$*, for some type* $\mathtt{S}$*, or, (2) an expression containing a subexpression* $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})\ \mathtt{as}\ \mathtt{T}$ *where* $\nvdash \mathtt{C} <: \mathtt{T}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})/\mathtt{self}]$*.*

PROOF. Combine Theorem A.10 and Theorem A.11. □