Constrained Types for Object-Oriented Languages

Vijay Saraswat[1], Nathaniel Nystrom[1], Jens Palsberg[2], and Christian Grothoff[3]

[1] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights NY 10598 USA, vsaraswa@us.ibm.com

[2] UCLA Computer Science Department, Boelter Hall, Los Angeles CA 90095 USA, palsberg@cs.ucla.edu

[3] Department of Computer Science, University of Denver, 2360 S. Gaylord Street, John Green Hall, Room 214, Denver CO, 80208 USA, christian@grothoff.org

Abstract. X10 is a modern object-oriented language designed for productivity and performance in concurrent and distributed systems, such as (heterogeneous) multicores and clusters. In this context, dependent types arise naturally: objects may be located at one of many places, arrays may be multidimensional, activities may be associated with one or more clocks, variables may be marked as shared or private following an ownership discipline, etc. A framework for dependent types offers 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 a simple, general framework for adding constraint-based dependent types to nominally typed OO languages such as Java, X10, and Scala. The framework is parametric on an underlying constraint system C. Classes and interfaces are associated with *properties* (= final instance fields). A type `C(:c)` names a class or interface `C` and a *constraint* `c` on the properties 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.

We present many examples to illustrate that many common OO idioms and OO type systems proposed recently can be naturally captured by constrained types: specifically we discuss types for places, aliases, ownership, arrays and clocks. We have implemented the type system in X10 and instantiated the framework with multiple constraint systems. We present a simple FJ extension, Constrained FJ, and establish fundamental properties such as type soundness. We compare this approach with relevant work in dependent types, specifically, DML, and outline many areas of future work.

In conclusion, we believe that constrained types offer a natural, simple, clean, and expressive extension to OO programming.

1 Introduction

X10 is a modern statically typed object-oriented (OO) language designed for high productivity in the high performance computing (HPC) domain [45]. Built essentially on the imperative sequential Java™ core, X10 introduces constructs for distribution and fine-grained concurrency (asynchrony, atomicity, ordering).

X10, like most OO languages is designed around the notion of objects as instances of *classes*. However, X10 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*, and mappings from index sets to places, known as *distributions*. X10 supports a rich algebra of operations over regions, distributions and arrays.

A central design goal of X10 is to rule out large classes of error by design. Thus, 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, `int[5]`, the type of all arrays of `int` of length `5`, `int[region(2)]`, the type of all `int` arrays over two dimensional regions, and `Obj!`, the type of all `Obj` 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) [45].

Thus, one needs a practical and usable framework for *dependent types* for concurrent, imperative, object-oriented languages. Dependent types are types parametrized by *values* [26] and have a rich and distinguished history in logic and functional programming languages (see e.g., [53, 34, 6, 7, 3]) and in the development of logical frameworks [11]. In the current setting, several questions must be answered: How are dependent types defined? What vocabulary of functions and predicates can be used in forming dependent types? Is this vocabulary extensible? How do dependent types interact with inheritance, method overloading and overriding, type casting? Are types checked statically? Can classes continue to be compiled separately in the presence of dependent types?

We believe these questions must be answered in a way that achieves certain desirable properties:

- **Ease of use.** The framework must be easy to use for practicing programmers. In particular, the syntax of types should be a simple and natural extension of existing syntax for types.
- **Flexibility.** The framework must permit the development of concrete, specific type systems tailored to the application area at hand, enabling a kind of pluggable type system [8]. Hence, the framework must be parametric in the kinds of expressions used in the type system.
- **Modularity.** The rules for type-checking must be specified once in a way that is independent of the particular vocabulary of operations used in the dependent type system.
- **Integration with OO languages.** The framework must work smoothly with nominal type systems found in Java-like languages. It must permit separate compilation.
- **Static checking.** The framework must permit mostly static type-checking. (The user should be able to escape the confines of static type-checking using dynamic casts, as is common for Java-like languages.)

1.1 Overview

In this paper we develop a general syntactic and semantic framework for *constrained types*, user-defined constraint-based types for modern class-based OO languages such

as Java, C♯ and X10. Our central insight is that a rich, user-extensible type system can be developed on top of predicates over the *immutable* state of objects. Such types statically capture many common invariants naturally arising in code. For instance, typically the shape of an array (number of dimensions, size of each dimension) is determined at runtime but fixed once the array is constructed. Thus, the shape of an array is part of its immutable state.

Properties. Our first step is to permit a class specification to specify *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 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.

Constraints. Given a class `C` with properties `T1 p1`, ..., `Tn pn`, a user may construct a potentially infinite family of types `C(:c)`, by using a *constraint* `c` on the properties of `C` and final variables in scope at the type. (`C` is called the *base class* of the type, and `c` its *condition* or *where clause*.) Constraints specify (possibly) partial information about the variables of interest.

Such a type represents a refinement of `C`—the set of all instances of `C` whose immutable state satisfies the condition `c`. The constraint is specified in terms of an underlying constraint system [46]—a pre-defined logical vocabulary of functions and predicates with algorithms for consistency and entailment. It 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)` represents the interior of a circle (for a class `point` with two `int` properties `x` and `y`). The type `C(:self != null)` represents all instances of `C`.[4] The type `int(:self==v)` represents a "singleton" type, an `int` is of this type only if it has the same value as `v`.

Example 1. Consider the class definition:

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

The type `List(:n==3)` is permissible and represents the set of all lists with length `3`. Similarly for `List(:n <= 41)` and even `List(:n * f() >= g())`.

Interface properties. Since properties play a central role in the specification of refinements of a type, it makes sense to permit interfaces to specify properties. Any class implementing the interface is required to define at least the same properties as the interface. It is an error for a class to inherit a property with the same name but different types from two different interfaces. Thus, an object `o` satisfies a type `I(:c)` if it is an instance of a class `C` implementing `I` and it satisfies `c`.

[4] When there is no ambiguity a property reference `self.x` may be abbreviated to `x`.

Properties are typed. 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 `this.region.rank=this.rank`. Similarly, an array has properties `dist`, `region`, and `rank` and appropriate constraints ensuring that the statically available information about them is consistent.[5]

In this way a class invariant (and constraint clauses in constrained types) may specify fairly rich constraints on the immutable portion of the object reference graph rooted at the current object, and utilizing objects at user-defined types.

Subtyping. Constrained types are naturally endowed 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`. Thus, the set of constrained types on a base type `C` inherits a lattice structure from the underlying constraint system. The maximal element is `C(:true)`, which is just `C` and the minimal element is the unsatisfiable constraint.

Constrained types may occur wherever normal types occur. In particular, they may be used to specify the types of (possibly mutable) local variables, properties, (possibly mutable) fields, arguments to methods, return types of methods, in casts etc. Note that final local variables as well as final parameters of methods may be used to define types. Specifically all the final parameters of a method are in scope at the return type of the method and can be used to construct the return type.

Example 2. A method that multiplies two matrices may be specified by:

```
class Matrix(int I, int J) {
  Matrix(I, arg.J) mul(final Matrix(:self.I==this.J) arg)
  {...}
  ...
}
```

Method overloading. Several design alternatives are possible for integrating method overloading, overriding, shadowing and obscuring [19] with dependent types. We discuss in more detail in Section 2. In summary, 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.

Invariants. For additional expressiveness, we permit the specification of a *class invariant*, a *where clause* [12] in the class definition. A where clause is a boolean expression on the parameters separated from the parameter list with a "`:`". The compiler ensures that all instances of the class created at runtime satisfy the where clause associated with the class.

[5] 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].

Where clauses may also be specified in argument lists of methods and constructors. They place an additional constraint on the actual arguments to the method or constructor.

For brevity, we write `C` as a type as well; it corresponds to the (vacuously) constrained type `C(:true)`. We also permit the syntax `C(t1,..., tk)` for the type `C (:x1 = t1, ..., xk = tk)` (assuming that the property list for `C` specifies the `k` properties `x1,..., xk`, and each term `ti` is of the correct type).

Thus, using the definition above, `List(n)` is the type of all lists of length `n`.

Example 3 (List). 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.

```
class List(int(:self >= 0) n) {
  Object head = null;
  List(n-1) tail = null;
  List(0)() { property(0); }
  List(1)(Object head) { this(head, new List());}
  List(tail.n+1)(Object head, List tail) {
    property(tail.n+1);
    this.head = head;
    this.tail = tail;
  }
  List(n+arg.n) append(final List arg) {
    return (n == 0)
      ? arg : new List(head, tail.append(arg));
  }
  List(n) rev() { return rev(new List()); }
  List(n+acc.n) rev(final List acc) {
    return (n == 0)
     ? acc : tail.rev(new List(head, acc));
  }
  List(:self.n <= this.n) filter(Predicate f) {
    if (n==0) return this;
    List(:self.n <=this.n-1) l = tail.filter(f);
    return (f.isTrue(head)) ?
     new List(head,l) : l;
  }
}
```

Intuitively, this definition states that a `List` has a `int` property `n`, which must be non-negative. The class has two fields that hold the head and tail of the list. The properties of a class are set through the invocation of `property(...);` (analogously to `super(...);`). Constructors have "return types" that can specify a where clause satisfied by the object being returned by the constructor. The compiler verifies that the constructor postcondition and the class invariant are implied by the `property` statement and any `super` calls in the constructor body imply

The second constructor returns a singleton list of length `1`. The third constructor returns a list of length `m+1`, where `m` is the length of the second argument. If an argument appears in the return type then the argument must be declared `final`. Thus,

the argument will point to the same object throughout the evaluation of the constructor body.

Final variables. The use of final variables (local variables, method arguments) in 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.

During type-checking, final variables are turned into symbolic variables—some fixed but unknown value—of the same type. Computation is performed in a constraint-based fashion on such variables.

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 knows how to process pieces of partial information about these logical variables in order to determine whether some constraint is entailed. At runtime, the same program variable will have a concrete value and will perform "arithmetic" (calculations) where the compiler performed "algebra" (symbolic analysis).

Query evaluation. Because OO languages permit arbitrary mutual recursion between classes—class `A` and `B` may have fields of type `B` and `A` respectively—the type/property graph may have loops. 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 `c` but also includes constraints that hold for all instances of `C`, as determined by the definition of `C`. Let us use the notation $rc(\mathtt{C})$ for the *real clause* of `C`. Since we consider only top-level classes, the only free variable in $rc(\mathtt{C})$ is `self`.

What is $rc(\mathtt{C},\mathtt{X})$ (we have drawn out `X` as the formal variable)? Consider a general class definition:

$$\mathtt{class}\ \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{x}} : c)\mathtt{extends}\ D(:d)\ \{\overline{M}\}$$

Clearly, from this we get:

$$rc(\mathtt{C},\mathtt{X}) \leftrightarrow \begin{array}{l}(\mathtt{c},\mathtt{d})[\mathtt{X}/\mathtt{this}] \wedge rc(\mathtt{D},\mathtt{X}) \wedge \\ rc(\mathtt{P}_1,\mathtt{X}.\mathtt{x}_1) \wedge \cdots \wedge rc(\mathtt{P}_k,\mathtt{X}.\mathtt{x}_k)\end{array}$$

That is, given a program P with classes $\mathtt{C}_1,\ldots,\mathtt{C}_k$, the set of real-clauses for $\mathtt{C}_1,\ldots,\mathtt{C}_k$ 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 $rc(\mathtt{C},\mathtt{X})$ 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.

In practice, many data-structures have non-cyclic dependency graphs. For such programs the real-clause can be computed quickly and only a bounded number of questions to the constraint-solver are generated during type-checking.

Existential Types. The use of constraints makes existential types very natural. Consider the return type of `filter` above—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.

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 where clause is solvable for each possible assignment of values to parameters. A parametrically consistent type has the property that its extension will never be empty.

Types are required to be parametrically consistent.

Example 4 (List, revisited). Consider a variation of List:

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

The type of the field `tail` is not parametrically consistent. There exists a value for its parameter `n`, namely, `0` for which the where clause `self != null => (self.n=this.n-1, self.n >= 0), self!=null` is not satisfiable.

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.

Runtime type casts. Constrained types may occur in a class cast `(T) o`. Code is generated to check at runtime that `o` satisfies `T`.

1.2 Claims

The paper presents a framework for integrating dependent types into Java-like OO languages by augmenting nominal types with constraints on properties of the type. This framework is the basis for the design of the dependent type system for X10. The design has been implemented in X10 version 1.0 (for a simple equality-based constraint system), available at `x10.sf.net`. The implementation is discussed in Section 4.

As in staged languages [29, 50], 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 ("for all possible values of method parameters") for dependent type-checking. There is no run-time constraint-solving. However, run-time casts to dependent types are permitted; these casts 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 natural and easy to use and useful. Many example programs have been written using dependent types and are available at `x10.sf.net`.

We claim that the design is flexible. It is parametric on the constraint system being used. We are planning on extending the current implementation to support multiple user-defined constraint systems, thereby supporting pluggable types. Dependent where clauses are also the basis for a general user-definable annotation framework we have implemented separately [32].

We claim the design is clean and modular. We present a simple core language CFJ, extending FJ [22] 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. We establish subject reduction and progress theorems.

Rest of this paper. The next section reviews related work. Section 2 fleshes out the syntactic and semantic details of the proposal, and presents a formal semantics and a soundness theorem. Section 3 works through a number of examples using a variety of constraint systems. The implementation of constrained types in X10 is described in Section 4. Section 5 conclude the paper with a discussion of future work.

1.3 Related work

Constraint-based type systems enjoy a long history. Mitchell [28] and Reynolds [44] developed the use of constraints for type inference and subtyping. Trifonov and Smith [51] proposed a type system where types are refined by subtyping constraints. Dependent types are not supported. Pottier [38, 40] presents a constraint-based type system for an ML-like language with subtyping.

HM(X) [49, 39, 41] is a constraint-based framework for Hindley–Milner style type systems. The framework is parameterized on the specific constraint system X; instantiating X yields extensions of the HM type system. Constraints in HM(X) are over types, not values.

Several systems have been proposed that refine types in a base type system through constraints. *Refinement types* [18] extend the Hindley–Milner type system with intersection, union, and constructor types, enabling specification and inference of more precise type information. *Conditional types* [1] extend refinement types to encode control-flow information in the types. Jones introduced *qualified types*, which permit types to be constrained by a finite set of predicates [23]. *Sized types* [21] annotate types with the sizes of recursive data structures. Sizes are linear functions of size variables. Size inference is performed using a constraint solver for Presburger arithmetic [43].

Our work is most closely related to DML, the extension of ML with dependent types. We discuss this in detail in the next section.

With hybrid type-checking [14, 15], types can be constrained by arbitrary boolean expressions. While typing is undecidable, dynamic checks are inserted into where the type-checker cannot accept or reject a program.

Singleton types [5, 48] are dependent types containing only one value. In Stone's formulation [48], $S(e:\tau)$ is the type of all values of type τ that are equal to e. Term equivalence is constructed so that type-checking is decidable. The singleton $S(e:\tau)$ can be encoded in CFJ as $\tau(:\texttt{self}{=}\lfloor e \rfloor)$, where $\lfloor e \rfloor$ lifts e to a constraint term.

Several languages—gbeta [13], Scala [33, 36], J& [31] and others [35, 34]—provide *path-dependent types*. For a final access path `p`, `p.type` in Scala is the singleton type containing the object `p`. In J& `p.class` is a type containing all objects whose run-time class is the same as `p`'s. Scala's `p.type` can be encoded in CFJ using an equality constraint `C(:self == p)`, where `C` is a supertype of `p`'s static type.

Cayenne [7] is a Haskell-like language with fully dependent types. There is no distinction between static and dynamic types. Type-checking is undecidable. There is no notion of datatype refinement as in DML.

Epigram [27, 3] is a dependently typed functional programming language based on a type theory with inductive families. Epigram does not have a phase distinction between values and types.

ESC/Java [16] 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. ESC/Java can enforce invariants on mutable state.

Pluggable and optional type systems were proposed by Bracha [8] and provide another means of specifying refinement types. Type annotations, implemented in compiler plugins, serve only to reject programs statically that might otherwise have dynamic type errors. CQual [17] extends C with user-defined type qualifiers. These qualifiers may be flow-sensitive and may be inferred. CQual supports only a fixed set of typing rules for all qualifiers. In contrast, the *semantic type qualifiers* of Chin, Markstrum, and Millstein [10] allow programmers to define typing rules for qualifiers in a meta language that allows type-checking rules to be specified declaratively. JavaCOP [4] is a pluggable type system framework for Java. Annotations are defined in a meta language that allows type-checking rules to be specified declaratively. JSR 308 [24] is a proposal for adding user-defined type qualifiers to Java.

DML Our work is most closely related to DML, [53], an extension of ML with dependent types. DML is also built parametrically on a constraint solver. Types are refinement types; they do not affect the operational semantics and erasing the constraints yields a legal ML program.

At a conceptual level the intuitions behind the development of DML and constrained types are similar. Both are intended for practical programming by mainstream programmers, both introduce a strict separation between compile-time and run-time processing, are parametric on a constraint solver, and deal with mutually recursive data-structures,

mutable state, and higher-order functions (encoded as objects in the case of constrained types). Both support existential types.

The most obvious distinction between the two lies in the target domain: DML is designed for functional programming, specifically ML, whereas constrained types are designed for imperative, concurrent OO languages. Hence technically our development of constrained types takes the route of an extension to FJ. But there are several other crucial differences as well.

First, DML achieves its separation by not permitting program variables to be used in types. Instead, a parallel set of (universally or existentially quantified) "index" variables have to be introduced. For instance the typing of the `app` operation on lists is provided by:

```
fun('a)
  append(nil, ys) = ys
 | append(cons(x, xs), ys) = cons(x, append(xs,ys))
where append <| {m:nat}{n:nat}
                'a list(m) * 'a list(n) -> 'a list(m+n)
```

in contrast with the direct embedded expression with constrained types:

```
class List(int(:self >= 0) n) {
   Object item;
   List(n-1) tail;
   List(n+a.n) app(final List a) {
     return n==0 ? a : new List(item, tail.app(a));}
   ...
}
```

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. (As we have seen permitting arbitrary type/property graphs may lead to undecidability.) Note that DML style constraints are easily encoded in constrained types.

Third, DML does not permit any runtime checking of constraints (dynamic casts).

2 Constrained types and their formalization

Our basic approach to introducing dependent types into class-based statically typed OO languages is to follow the spirit of generic types, but use values instead of types.

We permit the definition of a class `C` to specify a list of typed parameters, or *properties*, `(T1 x1, ..., Tk xk)` similar in syntactic structure to a method argument list. Each property in this list is treated as a `public final` instance field of the class. We also permit the specification of a *class invariant*, a *where clause* [12] in the class definition. A where clause is a boolean expression on the properties separated from the property list with a "`:`". The compiler ensures that all instances of the class created at runtime satisfy the where clause associated with the class. Thus, for instance, we may specify a class `List` with an `int length` property as follows:

```
class List(int length: 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 <= 41)` and even `List(:length * f() >= g())`.

Accordingly, a *constrained type* is of the form `C(:e)`, the name of a class or interface C, called the *base class*, followed by a where clause e, called the *condition*, a boolean expression on the properties of the base class. The denotation, or semantic interpretation, of such a type is the set of all instances of subclasses of the base class whose properties satisfy the condition. Clearly, for the denotation of a constrained type t to be non-empty the condition of t must be consistent with the class invariant, if any, of the base class of t. The compiler is required to ensure that the type of any variable declaration is non-empty.

For brevity, we write C as a type as well; it corresponds to the (vacuously) constrained type `C(:true)`. We also permit the syntax `C(t1,..., tk)` for the type `C(:x1 = t1, ..., xk = tk)` (assuming that the property list for C specifies the k properties `x1,..., xk`, and each term `ti` is of the correct type).

Constrained types naturally come equipped with a *subtyping structure*: type t_1 is a subtype of t_2 if the denotation of t_1 is a subset of t_2. This definition satisfies Liskov's Substitution Principle [25]), and implies that `C(:e1)` is a subtype of `C(:e2)` if `e1` implies `e2`. 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 `C1(:e1)` and `C2(:e2)` are considered equivalent if `C1` and `C2` are the same base type and `e1` and `e2` are equivalent when considered as logical expressions. Thus `C(:x*x=4)` and `C(:x=2 || x=-2)` are equivalent types.

2.1 Method and constructor preconditions

Methods and constructors may specify preconditions on their parameters as where clauses. For an invocation of a method (or constructor) to be type-correct, the associated where clause must be statically known to be satisfied. Note that the where clause may contain constraints on the properties of `this`. Thus the where clause may be used to specify that a method is *conditionally* available on some objects of the class and not others.

The return type of a method may also contain expressions involving the arguments to the method. However, we will require that any argument used in this way must be declared `final`, ensuring it is not mutated by the method body. For instance:

```
List(arg.length-1)
  tail(final List arg : arg.length > 0) {...}
```

will be a valid method declaration. It says that `tail` must be passed a non-empty list, and it returns a list whose length is one less than its argument.

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

The definition of a method declaration `m1` "having the same signature as" a method declaration `m2` involves identity of types. Two X10 types are defined to be identical iff they are equivalent. Two methods are said to have *the same signature* if (a) they have the same number of formal parameters, (b) for each parameter their types are equivalent, and (c) the constraints associated with their parameter list (if any) are equivalent. It is a compile-time error for there to be two methods with the same name and same signature in a class (either defined in that class or in a superclass).

A class `C` inherits from its direct superclass and superinterfaces all their methods visible according to the access restriction modifiers public/private/protected/(package) of the superclass/superinterfaces 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 the same signature. Methods are overridden on a signature-by-signature basis.

Dynamic method lookup does not take dependent type information into account, only the class hierarchy. 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. Examples are not difficult to construct.)

The current X10 compiler produces Java code. It further implements the restriction that no two methods for the same class can have the same signature after their constraints are erased. This simplifies implementation – no name mangling is needed to preserve the dependent type distinction in the generated Java code. However, this does cut down on the usefulness of constrained clauses for method dispatch.

2.2 Constructors for dependent classes

Like a method definition, a constructor may specify preconditions on its arguments and a postcondition on the value produced by the constructor.

Postconditions may be specified in a constructor declaration between the name of the class and the argument list of the constructor using a where clause. The where clause can reference only the properties of the class.

For instance, the nullary constructor for `List` ensures that the property `length` has the value `0`:

```
public 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.

2.3 Constraints

In this framework, 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.

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.

2.4 Extending dependent classes

A class may extend a constrained class, e.g. `class`$\mathtt{C}(\ldots)$ `extends` $\mathtt{D}(:\mathtt{d})\ldots$. 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 defind.

2.5 Dependent interfaces

Java does not allow interfaces to specify instance fields. Rather all fields in an interface are final static fields (constants).

X10 supports rich user-definable extensions to the type system by allowing the user of a type to construct new constrained types: new types that are predicates on the immutable state of the base type. For interfaces to support this extension, they must support user-definable properties, so that constrained types can be built over interfaces.

As with classes, an interface definition may specify properties in a list after the name of the interface. Similarly, an interface definition may specify a where clause in its property list. Methods in the body of an interface may have where clauses 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 specify a property with the same name, then they must all agree on the type of the property. The class must have a single property with the given name and type.

The general form of a class declaration is now:

```
class C(T1 x1, ..., Tk xk)
      extends B(:c)
      implements I1(:c1), ..., In(:cn) {...}
```

For such a declaration to type-check, it must be that the class invariant implies *inv*(I), c, where *inv*(I) is the invariant associated with interface I. Again, a constrained class or interface I is taken as shorthand for I(:true). 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.

2.6 Semantics

In this section we formalize a small fragment of X10 to illustrate the basic concepts behind constrained type-checking. In fact a very tiny language is chosen—a small extension of FJ with constrained types.

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 onl 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, 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.

We do model properties, constrained clauses, class invariants, where clauses in methods and constructors, and dependent type casts.

Constraint system. Constraints are assumed to be drawn from a fixed constraint system, $\mathcal{C}$, with inference relation $\vdash_{\mathcal{C}}$ [46]. All constraint systems are required to support the trivial constraint true, conjunction, existential quantification and equality on constraint terms. Constraint terms include (final) variables, the special variable self (which may occur only in constraints c which occur in a constrained type C(:c)), and field selections t.f.

We summarize here properties of constraint systems described in [46] that are needed for the proofs: constraint systems may be thought of as presented via an intuitionistic Gentzen proof system supporting identity; affine and exchange on the left; existential quantification and conjunction on the left and right; and closure under substitution of

Syntax for CFJ. The definitions are based on those in Featherweight Java [22].

$$\begin{array}{rl} \text{(Class)} & \mathtt{L} ::= \mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends\ T}\ \{\overline{\mathtt{M}}\} \\ \text{(Method)} & \mathtt{M} ::= \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c})\{\mathtt{return\ e;}\} \\ \text{(Expr)} & \mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}(\overline{\mathtt{e}}) \mid (\mathtt{T})\mathtt{e} \\ \text{(Type)} & \mathtt{S,T,U,Z} ::= \mathtt{C}(:\mathtt{d}) \end{array}$$

The *base type* of a type `C(:c)` (read as `C` *with* `c`) is `C`. We use the following shorthand for types: For a type `T` equal to `C(:c)`, we will write `S x; T` for `C(:S x; c)`, and `d,T` for `C(:d,c)`. Application of substitutions is extended to types by: $\mathtt{C}(:\mathtt{c})\theta = \mathtt{C}(:\mathtt{c}\theta)$.

CFJ subtyping judgment. We add a single rule to the rules of FJ:

$$\mathtt{C} \sqsubseteq \mathtt{C} \quad \frac{\mathtt{class\ C}(\ldots)\ \mathtt{extends\ D}(\ldots)\{\ldots\}}{\mathtt{C} \sqsubseteq \mathtt{D}} \quad \frac{\mathtt{C} \sqsubseteq \mathtt{D} \quad \mathtt{D} \sqsubseteq \mathtt{E}}{\mathtt{C} \sqsubseteq \mathtt{E}} \quad \frac{\mathtt{C} \sqsubseteq \mathtt{D}\ \ \sigma(\Gamma, \mathtt{C}(:\mathtt{c})\ \mathtt{x}) \vdash_{\mathcal{C}} \mathtt{d}[\mathtt{x}/\mathtt{self}]\ \ (\mathtt{x}\text{ fresh})}{\Gamma \vdash \mathtt{C}(:\mathtt{c}) \sqsubseteq \mathtt{D}(:\mathtt{d})}$$

(Whenever we state an assumption of the form "`x` is fresh" in a rule we mean it is not free in the consequent of the rule.)

CFJ typing judgment. We let Γ stand for multisets of type assertions, of the form $\mathtt{T\ x}$,[6] and constraints. Typing judgments are of the form $\Gamma \vdash \mathtt{S\ t}$ When Γ is empty, it is omitted.

Let `C` be a class declared as $\mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends\ D}(:\mathtt{d})\{\overline{\mathtt{M}}\}$. Let θ be a substitution and the type `T` be based on `C`. We define $inv(T,\theta)$ as the conjunction $\mathtt{c}\theta, \mathtt{d}\theta$ and (recursively) $inv(\mathtt{D},\theta)$. We bottom out with $inv(\mathtt{Object},\theta) = \mathtt{true}$. For a variable `x`, we use the shorthand $inv(\mathtt{C},\mathtt{x})$ to mean $inv(\mathtt{C},[\mathtt{x}/\mathtt{self}])$.

The definition of *mtype*(C,m) (the signature of a method named `m` in class `C`), *mbody*(C,m), (the body associated with method `m` in type `C`) and *fields*(C) (the sequence of fields and their types inherited or defined at `C`) is essentially as specified in FJ [22] with the difference that the method of a signature is taken to be of the form $\overline{\mathtt{S}}\ \overline{\mathtt{x}} : \mathtt{c} \to \mathtt{T}$. The variables `x` are permitted to occur in the types $\overline{\mathtt{S}}, \mathtt{T}$, and are considered bound, and subject to alpha-renaming. The definitions of *mtype*, *mbody*, *fields* are extended to apply to constrained types by ignoring the constraint. For a substitution θ we define $mtype(\mathtt{T},\mathtt{m},\theta)$ as the signature obtained by applying θ to $mtype(\mathtt{T},\mathtt{m})$, renaming bound variables as necessary. Similarly, for a substitution θ we define $fields(\mathtt{T},\theta)$ to be $\overline{S}\theta\ \overline{f}$, if the sequence of inherited and defined fields of the class underlying the type `T` is $\overline{S}\ \overline{f}$. We let $fields(\mathtt{T},\mathtt{x})$ stand for $fields(\mathtt{T},[\mathtt{x}/\mathtt{self}])$.

We define $\sigma(\Gamma)$ to be the set of constraints obtained from Γ by replacing each type assertion $\mathtt{C}(:\mathtt{d})\ \mathtt{x}$ in Γ with $\mathtt{d}[\mathtt{x}/\mathtt{self}], inv(C,x)$ and retaining any constraint in Γ.

$$\frac{\sigma(\Gamma, \mathtt{C}(:\mathtt{c})\ \mathtt{x}) \vdash_{\mathcal{C}} \mathtt{d}[\mathtt{x}/\mathtt{self}]}{\Gamma, \mathtt{C}(:\mathtt{c})\ \mathtt{x} \vdash \mathtt{C}(:\mathtt{d})\ \mathtt{x}}\ (\text{T-VAR}) \quad \frac{\Gamma \vdash \mathtt{T}_0\ \mathtt{e} \quad fields(\mathtt{T}_0,\mathtt{z}_0) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}_i \quad (\mathtt{z}_0\text{ fresh})}{\Gamma \vdash (\mathtt{T}_0\ \mathtt{z}_0; \mathtt{z}_0.\mathtt{f}_i = \mathtt{self}; \mathtt{U}_i)\ \mathtt{e}.\mathtt{f}_i}\ (\text{T-FIELD}) \quad \frac{\Gamma \vdash \mathtt{S\ e}}{\Gamma \vdash \mathtt{T}\ (\mathtt{T})\mathtt{e}}\ (\text{T-CAST})$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{T}_{0:n}\ \mathtt{e}_{0:n} & mtype(\mathtt{T}_0,\mathtt{m},\mathtt{z}_0) = \mathtt{Z}_{1:n}\ \mathtt{z}_{1:n} : \mathtt{c} \to \mathtt{S} \\ \Gamma, \mathtt{T}_{0:n}\ \mathtt{z}_{0:n} \vdash \mathtt{T}_{1:n} \sqsubseteq \mathtt{Z}_{1:n} & \sigma(\Gamma, \mathtt{T}_{0:n}\ \mathtt{z}_{0:n}) \vdash_{\mathcal{C}} \mathtt{c}\ \ (\mathtt{z}_{0:n}\text{ fresh}) \end{array}}{\Gamma \vdash (\mathtt{T}_{0:n}\ \mathtt{z}_{0:n}; S)\ \mathtt{e}_0.\mathtt{m}(\mathtt{e}_{1:n})}\ (\text{T-INVK}) \quad \frac{\begin{array}{ll} \Gamma \vdash \overline{\mathtt{T}}\ \overline{\mathtt{e}} & \theta = [\overline{\mathtt{f}}/\mathtt{this}.\overline{\mathtt{f}}]\ fields(C,\theta) = \overline{\mathtt{Z}}\ \overline{\mathtt{f}} \\ \Gamma, \overline{\mathtt{T}}\ \overline{\mathtt{f}} \vdash \overline{\mathtt{T}} \sqsubseteq \overline{\mathtt{Z}} & \sigma(\Gamma, \overline{\mathtt{T}}\ \overline{\mathtt{f}}) \vdash_{\mathcal{C}} inv(\mathtt{C},\theta) \end{array}}{\Gamma \vdash \mathtt{C}(:\overline{\mathtt{T}}\ \overline{\mathtt{f}}; \mathtt{self}.\overline{\mathtt{f}} = \overline{\mathtt{f}})\ \mathtt{new\ C}(\overline{\mathtt{e}})}\ (\text{T-NEW}$$

Method and class typing.

$$\frac{\overline{\mathtt{T}}\ \overline{\mathtt{x}}, \mathtt{C\ this}, \mathtt{c} \vdash \mathtt{S\ e}, \mathtt{S} \sqsubseteq \mathtt{T}}{\mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c})\{\mathtt{return\ e;}\}\ \text{OK in C}} \quad \frac{\overline{M}\ \text{OK in } C}{\mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends\ D}(:\mathtt{d})\ \{\overline{\mathtt{M}}\}\ \text{OK}}$$

Computation.

$$\frac{fields(C) = \overline{C}\ \overline{f}}{(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{f}_i \longrightarrow \mathtt{e}_i}\ (\text{R-FIELD}) \quad \frac{mbody(\mathtt{m},\mathtt{C}) = \overline{x}.\mathtt{e}_0}{(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow [\overline{d}/\overline{x}, \mathtt{new}\ C(\overline{e})/\mathtt{this}]\mathtt{e}_0}\ (\text{R-INVK}) \quad \frac{\vdash C \sqsubseteq T[\mathtt{new}\ C(\overline{\mathtt{d}})/\mathtt{self}]}{(\mathtt{T})(\mathtt{new\ C}(\overline{\mathtt{d}})) \longrightarrow \mathtt{new}\ C(\overline{\mathtt{d}})}\ (\text{R-CAST})$$

Congruence.

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0'.\mathtt{f}}\ (\text{RC-FIELD}) \quad \frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0'.\mathtt{m}(\overline{\mathtt{e}})}\ (\text{RC-INVK-RECV}) \quad \frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i',\ldots)}\ (\text{RC-INVK-ARG})$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{new\ C}(\ldots,\mathtt{e}_i',\ldots)}\ (\text{RC-NEW-ARG}) \quad \frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{(\mathtt{C})\mathtt{e}_0 \longrightarrow (\mathtt{C})\mathtt{e}_0'}\ (\text{RC-CAST})$$

Fig. 1. The system Constrained FJ

terms. We denote the application of the substitution $\theta = [\overline{\mathtt{t}}/\overline{\mathtt{x}}]$ to a constraint $\mathtt{c}$ by $\mathtt{c}[\overline{\mathtt{t}}/\overline{\mathtt{x}}]$.

$$
\begin{array}{rl}
\text{(C Term)} & \mathtt{t} ::= \mathtt{x} \mid \mathtt{self} \mid \mathtt{this} \mid \mathtt{t.f} \\
& \quad \mid \mathtt{new\ C}(\overline{\mathtt{t}}) \quad \mid \quad \mathtt{g}(\overline{\mathtt{t}}) \\
\text{(Constraint)} & \mathtt{c,d} ::= \mathtt{true} \mid \mathtt{p}(\overline{\mathtt{t}}) \\
& \quad \mid \mathtt{t} = \mathtt{t} \mid \mathtt{c,c} \mid \mathtt{T\,x;c}
\end{array}
$$

All constraint systems are required to satisfy: $\mathtt{new\ C}(\overline{\mathtt{t}}).\mathtt{f}_i = \mathtt{t}_i$ provided that $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ (for some sequence of types $\overline{\mathtt{T}}$).

Above, "," binds tighter than ";". We use the syntax `T x; c` for the constraint obtained by existentially quantifying the variable `x` of type `T` in `c`. `p` ranges over the collection of predicates supplied by the underlying constraint system, and `g` over the collection of functions.

Syntax. The syntax for the language is specified in Figure 1.

A type is taken to be of the form `C(:c)` where `C` is the name of a class or interface and `c` is a constraint; we say that `C` is the *base* of the type `C(:c)`.

A type assertion `C(:c) x` constrains the variable `x` to contain references to only those objects `o` that are instances of (subclasses of) `C` and for which the constraint `c` is true provided that occurrences of `self` in `c` are replaced by `o`. Thus in the constraint `c` of a constrained type `C(:c)`, `self` may be used to reference the object whose type is being specified. Note that `self` is distinct from `this`—`this` is permitted to occur in the clause of a type `T` only if `T` occurs in an instance field declaration or instance method declaration of a class; as usual, `this` is considered bound to the instance of the class to which the field or method declaration applies.

A *class declaration* $\mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends\ D}(:\mathtt{d})\ \{\overline{\mathtt{M}}\}$ is thought of as declaring a class `C` with the fields $\overline{\mathtt{f}}$ (of type $\overline{\mathtt{T}}$), a *declared class invariant* `c`, a *super-class invariant* `d` and a collection of methods $\overline{\mathtt{M}}$. The constraints `c` and `d` are true for all instances of the class `C` (this is verified in the rule for type-checking constructors, T-NEW). In these constraints, `this` may be used to reference the current object; `self` does not have any meaning and must not be used.

A *method declaration* $\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c})\{\ldots\}$ specifies the type of the arguments and the result, as usual. The method arguments $\overline{\mathtt{x}}$ may occur in the argument types $\overline{\mathtt{T}}$ and the return type $\mathtt{T}_0$. The constraint `c` specifies additional constraints on the arguments $\overline{\mathtt{x}}$ and `this` that must hold for a method invocation to be legal. Note that `self` does not make sense in `c` (no type is being defined), and must not occur in `c`.

Type judgments. Typing judgments are of the form $\Gamma \vdash \mathtt{T\ e}$ where Γ is a multiset of type assertions `T x` and constraints `c`.

T-VAR extends the identity rule ($\Gamma, x : C \vdash x : C$) of FJ to take into account the constraint entailment relation.

T-CAST encapsulates the three inference rules of FJ: T-UCAST, T-DCAST and T-SCAST for upwards cast, downwards cast, and "stupid" cast respectively.

In T-FIELD, we postulate the existence of a receiver object `o` of the given static type ($\mathtt{T}_0$). $\mathit{fields}(\mathtt{T}_0, \mathtt{o})$ is the set of typed fields for $\mathtt{T}_0$ with all occurrences of `this` replaced

by o. We record in the resulting constraint that $\mathtt{o}.\mathtt{f}_i = \mathtt{self}$.[7] This permits transfer of information that may have been recorded in $\mathtt{T}_0$ about the field $\mathtt{f}_i$.

Similarly, in T-INVK we postulate the existence of a receiver object o of the given static type. For any type T, object o of type T and method name m, let $mtype(\mathtt{T},\mathtt{m},\mathtt{o})$ be a copy of the signature of the method with this replaced by o. We establish (under the assumption that the formals ($\overline{\mathtt{z}}$) have the static type of the actuals)[8] that actual types are subtypes of the formal types, and the method constraint is satisfied. This permits us to record the constraint d on the return type, with the formal variables $\overline{\mathtt{z}}$ existentially quantified.[9]

In T-NEW, similarly, we establish that the static types of the actual arguments to the constructor are subtypes of the declared types of the field, and contain enough information to satisfy the class invariant, c. The declared types (and c) contain references to $\mathtt{this}.\overline{\mathtt{f}}$; these must be replaced by the formals $\overline{\mathtt{f}}$, which carry information about the static type of the actuals. Note that the object o we hypothesized in an analogous situation in T-INVK does not exist; it will exist on successful invocation of the constructor. The constrained clause of the new expression contains all the information that can be gleaned from the static types of the actuals by assigning them to the corresponding fields of the object being created.

Theorem 1 (Subject Reduction).
If $\Gamma \vdash T\ e$ *and* $e \longrightarrow e'$, *then for some type S,* $\Gamma \vdash S\ e'$ *and* $\Gamma \vdash S \sqsubseteq T$.

Let the normal form of expressions be given by *values*, i.e. expressions:

(Values) $\mathtt{v} ::= \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$

Theorem 2 (Progress). *If* $\vdash \mathtt{T}\ \mathtt{e}$, *then one of the following conditions holds:*

1. e *is a value* v,
2. e *contains a subexpression* $(\mathtt{T})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$ *such that* $\not\vdash \mathtt{C} \sqsubseteq \mathtt{T}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})/\mathtt{self}]$,
3. *there exists* $\mathtt{e}'$ *s.t.* $\mathtt{e} \longrightarrow \mathtt{e}'$.

Theorem 3 (Type Soundness).
If $\vdash \mathtt{T}\ \mathtt{e}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$, *with* $\mathtt{e}'$ *in normal form, then* $\mathtt{e}'$ *is either (1) a value* v *with* $\vdash \mathtt{S}\ \mathtt{v}$ *and* $\vdash \mathtt{S} \sqsubseteq \mathtt{T}$, *for some type* S, *or, (2) an expression containing a subexpression* $(\mathtt{T})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$ *where* $\not\vdash \mathtt{C} \sqsubseteq \mathtt{T}[\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})/\mathtt{self}]$.

Lemma 1 (Substitution Lemma). *Assume* $\Gamma \vdash \overline{\mathtt{A}}\ \overline{\mathtt{d}}$, $\Gamma \vdash \overline{\mathtt{A}} \sqsubseteq \overline{\mathtt{B}}$, *and* $\Gamma, \overline{\mathtt{B}}\ \overline{\mathtt{x}} \vdash \mathtt{T}\ \mathtt{e}$. *Then for some type* S *s.t.* $\Gamma \vdash \mathtt{S} \sqsubseteq \overline{\mathtt{A}}\ \overline{\mathtt{x}}; \mathtt{T}$ *it is the case that* $\Gamma \vdash \mathtt{S}\ \mathtt{e}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]$.

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

Lemma 3 (Body type). *If* $\mathrm{mtype}(\mathtt{T}_0,\mathtt{m}) = \overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c} \rightarrow \mathtt{S}$, *and* $mbody(\mathtt{m},\mathtt{T}_0) = \overline{\mathtt{x}}.\mathtt{e}$, *then for some* $\mathtt{U}_0$ *with* $\mathtt{T}_0 \sqsubseteq \mathtt{U}_0$, *there exists* $\mathtt{V} \sqsubseteq \mathtt{S}$ *such that* $\overline{\mathtt{T}}\ \overline{\mathtt{x}}, \mathtt{U}_0\ \mathtt{this} \vdash \mathtt{V}\ \mathtt{e}$

[7] A new name o is necessary to name this object since e cannot be used. Arbitrary term expressions e are not permitted in constraints; the functions used in e may not be known to the constraint system, and e may have side-effects.

[8] This is stronger than assuming $\overline{\mathtt{Z}}$.

[9] Recall that the $\overline{\mathtt{z}}$ may occur in d but must not occur in a type in the calling environment; hence they must be existentially quantified in the resulting constraint.

2.7 Erasure

Constrained types in CFJ are a form of *refinement type* [18]. If constraints are erased from a well-typed program, the resulting program will behave identically to the unerased program except that the unerased program might be unable to take a step on a cast.

Let $[\![\mathtt{e}]\!]$ be the erasure of $\mathtt{e}$ defined as follows:

$$\begin{aligned}
[\![\mathtt{x}]\!] &= \mathtt{x} \\
[\![\mathtt{e.f}]\!] &= [\![\mathtt{e}]\!].\mathtt{f} \\
[\![\mathtt{e.m}(\overline{\mathtt{e}})]\!] &= [\![\mathtt{e}]\!].\mathtt{m}(\overline{[\![\mathtt{e}]\!]}) \\
[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!] &= \mathtt{new\ C}(\overline{[\![\mathtt{e}]\!]}) \\
[\![(\mathtt{C}(:\mathtt{c}))\ \mathtt{e}]\!] &= (\mathtt{C}(:\mathtt{true}))\ \overline{[\![\mathtt{e}]\!]}
\end{aligned}$$

Theorem 4 (Erasure).
If $\vdash \mathtt{C}(:\mathtt{c})\ \mathtt{e}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{v}$*, then* $\vdash \mathtt{C}(:\mathtt{true})\ [\![\mathtt{e}]\!]$ *and* $[\![\mathtt{e}]\!] \longrightarrow^* [\![\mathtt{v}]\!]$.

3 Applied constrained calculi

The following section presents examples using several different constraint systems. Many of these constraint systems are more expressive than the constraint system implemented in the current X10 compiler and have not (yet) been implemented.

In the following we will use the shorthand $\mathtt{C}(\overline{\mathtt{t}} : \mathtt{c})$ for the type $\mathtt{C}(: \overline{\mathtt{f}} = \overline{\mathtt{t}}, \mathtt{c})$ where the declaration of the class $\mathtt{C}$ is $\mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c}) \ldots$ Also, we abbreviate $\mathtt{C}(\overline{\mathtt{t}} : \mathtt{true})$ as $\mathtt{C}(\overline{\mathtt{t}})$. Finally, we use the shorthand $\mathtt{T\ x} = \mathtt{t};\ \mathtt{c}$ for the constraint $\mathtt{T\ x};\ \mathtt{x} = \mathtt{t};\ \mathtt{c}$.

3.1 AVL trees and red–black trees

AVL trees and red-black trees can be modeled so that the data structure invariant is enforced statically.

```
class AVLTree(int(:self >= 0) height) {...}
class Leaf(Object key) extends AVLTree(0) {...}
class Node(Object key, AVLTree l, AVLTree r
         : int d=l.height-r.height; -1 <= d, d <= 1)
    extends AVLTree(max(l.height,r.height)+1) {...}
```

Red–black trees may be modeled similarly. Such trees have the invariant that (a) all leaves are black, (b) each non-leaf node has the same number of black nodes on every path to a leaf (the black height), (c) the immediate children of every red node are black.

```
class RBTree(int blackHeight) {...}
class Leaf extends RBTree(0) { int value; ... }
class Node(boolean isBlack,
           RBTree(:this.isBlack || isBlack) l,
           RBTree(:this.isBlack || isBlack,
                   blackHeight=l.blackHeight) r)
    extends RBTree(l.blackHeight+1) { int value; ... }
```

3.2 Array bounds

Xi and Pfenning proposed using dependent types for eliminating array bounds checks [52]. In CFJ, an array of type `T[]` indexed by (signed) integers can be modeled as a class with the following signature:[10]

```
interface Array<T>(int(:self >= 0) length) {
  T get(int(:0 <= self, self < this.length) i);
  void set(int(:0 <= self, self < this.length) i, T v);
}
```

Bounds can be checked using a constraint system based on Presburger arithmetic [43]. Constraint terms include integer constraints, scalar multiplication, and addition; constraints include inequalities:

(C Term) `t` ::= `n` | `n*t` | `t+t` | ...
(Constraint) `c` ::= `t <= t` | ...

3.3 Region-based arrays

X10 takes another approach to ensuring array bounds violations do no occur. Following ZPL [9], arrays in X10 are defined over sets of n-dimensional *index points* called *regions* [20]. 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`.

Constrained types ensure 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 and have the following syntax:

(Constraint) $\texttt{c} ::= \texttt{r} \subseteq \texttt{r} \mid \ldots$
(Region) $\texttt{r} ::= \texttt{t} \mid [\texttt{b}_1\texttt{:}\texttt{d}_1,\ldots,\texttt{b}_k\texttt{:}\texttt{d}_k] \mid \texttt{r|r} \mid \texttt{r\&r} \mid \texttt{r-r} \mid \texttt{r+p}$
(Point) $\texttt{p} ::= \texttt{t} \mid [\texttt{b}_1,\ldots,\texttt{b}_k]$
(Integer) $\texttt{b,d} ::= \texttt{t} \mid \texttt{n}$

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`.

For example, the code in Figure 2 performs a successive over-relaxation [42] of an $n \times n$ matrix `G`. The type-checker establishes that the `region` property of the point `ij` (line 17) is `inner & [i:i,d1min:d1max]`, and that this region is a subset of `outer`, the region of the array `G`.

3.4 Place types

This example is due to Satish Chandra. We wish to specify a balanced distributed tree with the property that its right child is always at the same place as its parent, and once the left child is at the same place then the entire subtree is at that place. (Recall that in

[10] For this example, we assume generics are supported.

```
1  point NORTH = new point(1,0);
2  point WEST  = new point(0,1);
3  void sor(double omega, double[.] G, int iter) {
4    region(:self=G.region) outer = G.region;
5    region(:self⊆outer) inner =
6      outer & (outer-NORTH) & (outer+NORTH)
7            & (outer-WEST)  & (outer+WEST);
8    region d0 = inner.project(0);
9    region d1 = inner.project(1);
10   if (d1.size() == 0) return;
11   int d1min = d1.min()[0];
12   int d1max = d1.max()[0];
13   for (point[off] : [1:iter*2]) {
14     int red_black = off % 2;
15     finish foreach (point[i]: d0) {
16       if (i % 2 == red_black) {
17         for (point ij: inner & [i:i,d1min:d1max]) {
18           G[ij] = omega / 4.
19                 * (G[ij-NORTH] + G[ij+NORTH]
20                  + G[ij-WEST]  + G[ij+WEST])
21                 * (1. - omega) * G[ij];
22         }
23       }
24     }
25   }
26 }
```

Fig. 2. Successive over-relaxation with regions

X10 every reference class has a field `loc` of type `place` which specifies the location at which this object is located.)

The desired property may be specified thus:

```
class Tree(boolean localLeft) extends Object {
  Tree(:this.localLeft => (loc=here,self.localLeft)) left;
  Tree(:loc=here) right);
  ...
}
```

Note that the consequent of the condition for `left` sets `self.localLeft`. This ensures, recursively, that the entire left subtree will be located at the same place.

3.5 Clocked types

Clocks are barriers that are adapted to a context where activities may be dynamically created, and are designed so that all clock operations are determinate.

For each arity n, we introduce a *Gentzen predicate* $\texttt{clocked}(\overline{\texttt{t}})$. A k-ary Gentzen predicate a satisfies the property that $a(t_1,\ldots,t_k) \vdash a(s_1,\ldots,s_n)$ iff $k = n$ and $t_i = s_i$ for $i \leq k$.

Such a `clocked` atom is added to the context by an `clocked async`:

$$\frac{\Gamma, \texttt{clocked}(\overline{\texttt{v}}) \vdash \texttt{T e}}{\Gamma \vdash \texttt{T async clocked}(\overline{\texttt{v}})\texttt{e}}$$

A programmer can require that a method may be invoked only if the invoking activity is registered on the clocks $\overline{\texttt{k}}$ by adding a `clocked` clause. The rule for method elaboration and method invocation then change:

$$\frac{\overline{\texttt{T}}\ \overline{\texttt{x}}, \texttt{C this}, \texttt{c}, \texttt{clocked}(\overline{\texttt{k}}) \vdash \texttt{S e}, \texttt{S} \sqsubseteq \texttt{T}}{\texttt{T m}(\overline{\texttt{T}}\,\overline{\texttt{x}} : \texttt{c})\texttt{clocked}(\overline{\texttt{k}})\{\texttt{return e;}\}\ \text{OK in C}}$$

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{T}_{0:n}\ \texttt{e}_{0:n} \\ \mathit{mtype}(\texttt{T}_0, \texttt{m}, \texttt{z}_0) = \texttt{Z}_{1:\texttt{n}}\ \texttt{z}_{1:\texttt{n}} : \texttt{c}, \texttt{clocked}(\overline{\texttt{k}}) \to \texttt{S} \\ \Gamma, \texttt{T}_{0:n}\ \texttt{z}_{0:n} \vdash \texttt{T}_{1:n} \sqsubseteq \texttt{Z}_{1:n} \\ \sigma(\Gamma, \texttt{T}_{0:n}\ \texttt{z}_{0:n}) \vdash_{\mathcal{C}} \texttt{c} \quad (\texttt{z}_{0:n}\ \text{fresh}) \\ \Gamma \vdash \texttt{clocked}(\overline{\texttt{k}})\end{array}}{\Gamma \vdash (\texttt{T}_{0:n}\ \texttt{z}_{0:n}; S)\ \texttt{e}_0.\texttt{m}(\texttt{e}_{1:\texttt{n}})}\ (\text{T-INVK})$$

3.6 Activity-local objects

Parallelism in X10 is supported through lightweight asynchronous *activities*, created by `async` statements. It is often useful to restrict objects so that they are *local* to a particular activity. A local object may be accessed only by the activity that created it or by an ancestor of that activity. Local objects are declared and created by qualifying their type with `local`:

```
local C o = new local C();
```

To encode local objects in CFJ, we add an `activity` property to objects:

```
class Object(Activity activity) { ... }
```

where `Activity` has a possibly null `parent` property:

```
class Activity(Activity parent) { ... }
```

To track the current activity (z), we augment typing judgments as follows:

$$\mathsf{z};\ \Gamma \vdash \mathtt{T}\ \mathtt{e}$$

where $\mathtt{Activity}(\mathsf{z}')\ \mathsf{z} \in \Gamma$. When the current activity is z, we encode the type `local C` as $\mathtt{C}(\mathsf{z})$.

Spawning a new activity with an `async` statement introduces a fresh activity z':

$$\frac{\mathsf{z}';\ \Gamma,\ \mathtt{Activity}(\mathsf{z})\ \mathsf{z}' \vdash \mathtt{T}\ \mathtt{e} \quad (\mathsf{z}' \text{ fresh})}{\mathsf{z};\ \Gamma \vdash \mathtt{T}\ (\mathtt{async}\ \mathtt{e})}$$

The rule T-FIELD is strengthened to require that reads only be performed on objects whose `activity` property is a descendant of the current activity.

$$\frac{\begin{array}{l}\mathsf{z};\ \Gamma \vdash \mathtt{T}_0\ \mathtt{e} \\ \mathit{fields}(\mathtt{T}_0, \mathsf{z}_0) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}_i \qquad\qquad (\mathsf{z}_0 \text{ fresh}) \\ \mathsf{z};\ \Gamma \vdash \mathtt{T}_0 \sqsubseteq \mathtt{C}(:\mathtt{activity} = \mathsf{z}')\ \Gamma \vdash \mathsf{z} \text{ spawns } \mathsf{z}'\end{array}}{\mathsf{z};\ \Gamma \vdash (\mathtt{T}_0\ \mathsf{z}_0; \mathsf{z}_0.\mathtt{f}_i = \mathtt{self}; \mathtt{U}_i)\ \mathtt{e}.\mathtt{f}_i}$$

where the spawns relation is defined as follows:

$$\Gamma \vdash \mathsf{z} \text{ spawns } \mathsf{z}$$

$$\frac{\Gamma \vdash \mathsf{z}_1 \text{ spawns } \mathsf{z}_2 \quad \Gamma \vdash \mathsf{z}_2 \text{ spawns } \mathsf{z}_3}{\Gamma \vdash \mathsf{z}_1 \text{ spawns } \mathsf{z}_3}$$

$$\frac{\Gamma \vdash \mathsf{z}_2.\mathtt{parent} = \mathsf{z}_1}{\Gamma \vdash \mathsf{z}_1 \text{ spawns } \mathsf{z}_2}$$

4 Implementation

Constrained types are implemented in the X10 language. Constraints in X10 are conjunctions of equalities over immutable side-effect-free expressions.

The X10 compiler is implemented as an extension of Java using the Polyglot compiler framework [30]. Expressions used in constrained types are type-checked as normal non-dependent X10 expressions; no constraint solving is performed. A separate compiler pass generates and solves constraints via an ask–tell interface [47]. If constraints cannot be solved, an error is reported.

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, dependent types may be used in casts and `instanceof` expressions. Because the constraint syntax in X10 is a subset of the X10 expression syntax, run-time tests of constrained types are translated to Java by evaluating the constraint with `self` bound to the expression being tested. For examples, casts are translated as:

```
⟦(C(:c) e⟧ =
  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.

5 Conclusion and Future work

We have presented a simple design for constrained types in Java-like languages. 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. We have formalized constrained types in a sound extension of FJ, Constrained FJ. Several examples of constrained types were presented. Constrained types have been implemented in X10 and used for place types, clocked types, and array types.

In future work, we plan to investigate optimizations (such as array bounds check elimination) 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.

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. Subtyping with singleton types. In *CSL '94: Selected Papers from the 8th International Workshop on Computer Science Logic*, volume 933 of *LNCS*, pages 1–15, London, UK, 1995. Springer-Verlag.
6. David Aspinall and Martin Hofmann. *Dependent Types*, chapter 2. In Pierce [37], 2004.
7. 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.
8. Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.
9. 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.
10. 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.
11. Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.
12. Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 156–168, Austin, TX, October 1995.
13. Erik Ernst. *gbeta: A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.
14. Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd Annual Symposium on Principles of Programming Languages (POPL'06)*, pages 245–256, 2006.
15. 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.
16. 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.
17. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 29th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–12. ACM Press, June 2002.
18. Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, June 1991.
19. J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2006.
20. Christian Grothoff, Jens Palsberg, and Vijay Saraswat. Safe arrays via regions and dependent types. Technical Report RC23911, IBM T.J. Watson Research Center, 2006.
21. 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.

22. 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.
23. Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.
24. JSR 308: Annotations on Java types. `http://jcp.org/en/jsr/detail?id=308`.
25. Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(1):1811–1841, November 1994.
26. Per Martin-Löf. *A Theory of Types*. 1971.
27. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
28. 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.
29. Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, 1992.
30. 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.
31. Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software extension. In *Proceedings of the 2006 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 21–36, Portland, OR, October 2006.
32. Nathaniel Nystrom and Vijay Saraswat. An annotation and compiler plugin system for X10. Technical Report RC24198, IBM T.J. Watson Research Center, 2007.
33. 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. Technical report, École Polytechnique Fédérale de Lausanne, June 2004. `http://scala.epfl.ch/docu/files/ScalaOverview.pdf`.
34. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, July 2003.
35. Martin Odersky and Christoph Zenger. Nested types. In *8th Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.
36. Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA05*, pages 41–57, San Diego, CA, USA, October 2005.
37. Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.
38. François Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, 1996.
39. François Pottier. A semi-syntactic soundness proof for HM(X). Technical Report RR 4150, INRIA, March 2001.
40. François Pottier. Simplifying subtyping constraints, a theory. *Information and Computation*, 170(2):153–183, November 2001.
41. François Pottier and Didier Rémy. *The Essence of ML Type Inference*, chapter 10. In Pierce [37], 2004.
42. 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).

43. William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 4–13, 1991.
44. John C. Reynolds. Three approaches to type structure. In *Proceedings of TAPSOFT/CAAP 1985*, volume 185 of *LNCS*, pages 97–138. Springer-Verlag, 1985.
45. V. Saraswat et al. Report on the programming language X10. Technical report, IBM T.J. Watson Research Center, 2006.
46. Vijay Saraswat. The category of constraint systems is Cartesian closed. In *LICS '92*, pages 341–345, 1992.
47. Vijay Saraswat. *Concurrent Constraint Programming*. Doctoral Dissertation Award and Logic Programming. MIT Press, 1993.
48. Christopher A. Stone. *Singleton Types and Singleton Kinds*. PhD thesis, Carnegie–Mellon University, August 2000. Also availalble as CMU technical report CMU-CS-00-153.
49. 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.
50. 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.
51. Valery Trifonov and Scott Smith. Subtyping constrained types. In *Third International Static Analysis Symposium (SAS)*, number 1145 in LNCS, pages 349–365, 1996.
52. 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.
53. 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.