Constrained Types for Object-Oriented Languages

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

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

Radha Jagadeesan [†]
rjagadeesan@cs.depaul.edu

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, such as (heterogenous) 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 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 (for a simple equality-based constraint system) in X10 1.0 (available at `x10.sf.net`). We present a simple FJ extension, `Constrained FJ`, and establish fundamental properties such as type soundess. 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 [23]. 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.

In designing a static type system for X10 a central problem arises. It becomes necessary to permit types, such as `region(2)`, the type of all two-dimensional regions, `int[5]`, the type of all arrays of `int` of length 5, and `int[region(2)]`, the type of all `int` arrays over two dimensional regions. The underlying general idea is that of *dependent types*: types parametrized by *values* [**?**, 28].

XXX emph pluggable nature of the constraint system.

In this paper we develop a general syntactic and semantic framework for *constrained types*, user-defined dependent types in the context of 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. Given a single dependently typed class or interface `C`, a user may construct a potentially infinite family of types `C(:c)` where `c` is a predicate on the immutable state.

In designing this framework, we had the following criteria:

- **Ease of use.** The framework must be easy to use for practicing programmers.
- **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 [2]. 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, and must permit separate compilation.
- **Static checking.** The framework must permit mostly static type-checking.

1.1 Overview

XXX Overview of our design.

A *constrained type* `C(:c)` is a classification of the object type `C` using a constraint `c` on properties of the type.

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

[†] School of CTI, DePaul University, 243 S. Wabash Avenue, Chicago IL 60604 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

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* [6] 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. 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 which satisfy the boolean expression. Thus, `List(:length == 3)` is a permissible type, as are `List(:length <= 41)` and even `List(:length * f() >= g())`. If `C` has no properties, the only type that can be constructed is the type `C`.

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.

EXAMPLE 1.1. *The type* `C(:self != null)` *is true only for non-null values of type* `C`. *The type* `int(:self==v)` *represents a "singleton" type, an int is of this type only if it has the same value as* `v`.

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`.

1.2 An example

Figure 1 shows a `List` class. 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 first constructor `List()` has a postcondition that signifies that it returns a list of length 0. The constructor body contains a `property` statement that initializes the `length` property to `0`. 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.

Functions that append one list to another or that reverse a list can be defined quite naturally. In both cases the size of the list returned is known statically (as a function of the size of the list and the argument to the method). The example also illustrates a method `filter` which returns a list whose length cannot be known statically since it depends on the argument predicate `f`; the list length can be bounded, however.

1.3 Claims

XXX Claims of this paper.

As in staged languages [14, 26], 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.

We outline the design for a compiler that can use an extensible constraint-solver for type-checking. This design is implemented in the `X10` compiler. No extension of an underlying virtual machine is necessary, except as may be useful in improving efficiency (for example, eliminating array bounds checks). The compiler translates source programs into target programs without dependent types but with `assume` and `assert` statements. A general constraint propagator that depends only on the operational semantics of the language and is constrained on the underlying constraint solver may be run on the program in order to eliminate branches and `asserts` forced by the assumptions. If all `asserts` cannot be eliminated at compile-time, some residual constraint-checking (*not* solving) may need to be performed at runtime. XXX contrast with hybrid type checking.

Rest of this paper. The next section reviews related work. Section 2 fleshes out the syntactic and semantic details of the proposal. Formal semantics and a soundness theorem are presented in Section 2.6. Section 3 works through a number of examples using a variety of constraint systems. Sections 5 and 5 conclude the paper with a discussion of future work.

1.4 Related work

Constraint-based type systems enjoy a long history. Mitchell [13] and Reynolds [**?**] developed the use of constraints for type inference and subtyping. Trifonov and Smith [27] proposed a type system where types are refined by subtyping constraints. Dependent types are not supported. Pottier [18, 20] presents a constraint-based type system for an ML-like language with subtyping.

HM(X) [25, 19, 21] 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. XXX HM(X) introduced *term constraint systems*; constraints in CFJ are term constraints?

Several systems have been proposed that refine types in a base type system through constraints. *Refinement types* [**?**] extend the Hindley–Milner type system with intersection, union, and constructor types, enabling specification and inference of more precise type information. *Conditional types* [**?**] 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 [11]. *Sized types* [**?**] 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 [**?**].

Xi and Pfenning [28] describe an extension of ML with dependent types, DML(C), in which types are indexed by constraints from a domain C that is a parameter to the type system. Types in DML are refinement types; they do no affect the operational semantics and erasing the constraints yields an ML program. CFJ permits

```
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(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;
    if (f.isTrue(head)) {
      List l = tail.filter(f);
      return new List(l+1)(head, l);
    } else {
      return tail.filter(f);
    }
  }
}
```

Figure 1. List example

casts and run-time tests of constraints. Like CFJ, DML supports existenial dependent types.

With hybrid type-checking [**?**, **?**], 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 [**?**, **?**] are dependent types containing only one value. In Stone's formulation [**?**], $S(e:\tau)$ 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 [**?**], Scala [**?**, **?**], J& [**?**] and others [**?**, **?**]—provide *dependent path 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. These types can be encoded in CFJ similarly to self types using T-CONSTR, as described in Section 3.

Cayenne [**?**] is a Haskell-like langauge 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 [**?**] is a dependently typed functional programming language.

ESCJava [8] and Spec♯ [**?**] allow programmers to write invariants to be enforced statically by the compiler.

JSR 308, Javari

2. Constrained FJ

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* [6] 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 which satisfy the boolean expression. Thus, `List(:length == 3)` is a permissible type, as are `List(:length <= 41)` and even `List(:length * f() >= g())`. If `C` has no properties, the only type that can be constructed is the type `C`.

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

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 [12]), 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.

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. 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.

2.1.1 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.

2.2 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 colored 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 which 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.

2.3 Extending dependent classes

A class may extend a constrained class.

*MetaNote: This should be standard. A class definition may extend a dependent super class, e.g. class Foo(int i) extends Fum(i*i) { ... }. The expressions in the actual parameter list for the super class may involve only the properties of the class being defined. The intuition is that these parameters are analogous to explicit arguments that must be passed in every super-constructor invocation.*

2.4 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(:e)
      implements I1(:e1), ..., In(:en) {...}
```

For such a declaration to type-check, it must be that the class invariant implies `inv(I) && e`, 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.

2.5 XXX more stuff

CFJ with field assignments.

Discussion of language design issues

– how should method resolution be done in the presence of constrained types?

– conditional fields. – recursive definitions of predicates in the constraint language through the use of CLP.

Constraint system (generic presentation). Design is constraint-system agnostic.

Principal clause

2.6 Semantics

Syntax. The syntax for the language is specified in Figure 5.

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)`.

$$
\begin{array}{rrcl}
\text{(Class)} & \mathrm{L} & ::= & \texttt{class C}(\overline{\texttt{T f}} : \texttt{c})\ \texttt{extends T}\ \{\overline{\texttt{M}}\} \\
\text{(Method)} & \mathrm{M} & ::= & \texttt{T m}(\overline{\texttt{T x}} : \texttt{c})\{\texttt{return e;}\} \\
\text{(Expr)} & \mathrm{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid \texttt{(T)e} \\
\\
\text{(C Terms)} & \mathrm{t} & ::= & \texttt{x} \mid \texttt{self} \mid \texttt{this} \mid \texttt{t.f} \mid \texttt{new C}(\overline{\texttt{t}}) \\
\text{(Constraint)} & \mathrm{c,d} & ::= & \texttt{true} \mid \texttt{a} \mid \texttt{t} = \texttt{t} \mid \texttt{c,c} \mid \texttt{T x;c} \\
\text{(Type)} & \mathrm{S,T,U} & ::= & \texttt{C(: d)}
\end{array}
$$

Figure 2. Syntax for CFJ

We let Γ stand for multisets of type assertions, of the form $T\,x$, and constraints. We define $\sigma(\Gamma)$ to be the set of constraints obtained from Γ by replacing each type assertion $\texttt{C(: d) x}$ with $\texttt{d}[\texttt{x}/\texttt{self}]$.

$$
\texttt{C} \sqsubseteq \texttt{C} \qquad \frac{\texttt{class C}(\ldots)\ \texttt{extends D}(\ldots)\{\ldots\}}{\texttt{C} \sqsubseteq \texttt{D}}
$$

$$
\frac{\texttt{C} \sqsubseteq \texttt{D} \quad \texttt{D} \sqsubseteq \texttt{E}}{\texttt{C} \sqsubseteq \texttt{E}} \qquad \frac{\texttt{C} \sqsubseteq \texttt{D} \quad \sigma(\Gamma, \texttt{C(: c) x}) \vdash_{\mathcal{C}} \texttt{d}[\texttt{x}/\texttt{self}]\ \ (\texttt{x new})}{\Gamma \vdash \texttt{C(: c)} \sqsubseteq \texttt{D(: d)}}
$$

Figure 3. CFJ subtyping judgement

Let C be a class declared as $\texttt{class C}(\overline{\texttt{T f}} : \texttt{c})\ \texttt{extends D(: d)}\{\overline{\texttt{M}}\}$. Let θ be a substitution and the type T be based on C. We define $inv(T, \theta)$ as the conjunction $\texttt{c}\theta, \texttt{d}\theta$ and (recursively) $inv(\texttt{D}, \theta)$. We bottom out with $inv(\texttt{Object}, \theta) = \texttt{true}$. If the sequence of inherited and defined fields of the class underlying a type T are $\overline{S}\ \overline{f}$, we define $fields(\texttt{T}, \theta)$ to be $\overline{S}\theta\ \overline{f}$.

$$
\frac{\sigma(\Gamma, \texttt{C(: c) x}) \vdash_{\mathcal{C}} \texttt{d}[\texttt{x}/\texttt{self}]}{\Gamma, \texttt{C(: c) x} \vdash \texttt{C(: d) x}}\ \text{(T-Var)}
$$

$$
\frac{\Gamma \vdash \texttt{T}_0\ \texttt{e} \quad fields(\texttt{T}_0, \texttt{z}_0) = \overline{\texttt{U}}\ \overline{\texttt{f}}_i \quad (\texttt{z}_0 \text{ fresh})}{\Gamma \vdash (\texttt{T}_0\ \texttt{z}_0; \texttt{z}_0.\texttt{f}_i = \texttt{self;U}_i)\ \texttt{e.f}_i}\ \text{(T-Field)} \qquad \frac{\Gamma \vdash \texttt{S e}}{\Gamma \vdash \texttt{T (T)e}}\ \text{(T-Cast)}
$$

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

$$
\frac{\begin{array}{ll}\Gamma \vdash \overline{\texttt{T}}\ \overline{\texttt{e}} \quad \theta = [\overline{\texttt{f}}/\texttt{this}.\overline{\texttt{f}}] & fields(C, \theta) = \overline{\texttt{Z}}\ \overline{\texttt{f}} \\ \Gamma, \overline{\texttt{T}}\ \overline{\texttt{f}} \vdash \overline{\texttt{T}} \sqsubseteq \overline{\texttt{Z}} & \sigma(\Gamma, \overline{\texttt{T}}\ \overline{\texttt{f}}) \vdash_{\mathcal{C}} inv(\texttt{C}, \theta)\end{array}}{\Gamma \vdash \texttt{C(: } \overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; self.}\overline{\texttt{f}} = \overline{\texttt{f}}\texttt{)}\ \texttt{new C}(\overline{\texttt{e}})}\ \text{(T-New)}
$$

Method and Class Typing.

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

Figure 4. CFJ type judgement

Computation:

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

Congruence:

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

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

Figure 5. Reduction rules for Constrained FJ

Constraints are assumed to be drawn from a fixed constraint system, $\mathcal{C}$, with inference relation $\vdash_{\mathcal{C}}$. 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`. Finally, we assume that constraints are closed under variable substitution. We denote the application of the substitution $\theta = [\overline{t}/\overline{x}]$ to a constraint c by $c[\overline{t}/\overline{x}]$. Application of substitutions is extended to types by: $\mathtt{C}(:\mathtt{c})\theta = \mathtt{C}(:\mathtt{c}\theta)$.

All constraint systems are required to satisfy:

$$\mathtt{new}\ \mathtt{C}(\overline{\mathtt{t}}).\mathtt{f}_i = \mathtt{t}_i$$

provided that $fields(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ (for some sequence of types $\overline{\mathtt{T}}$).

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* $\mathtt{class}\ \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends}\ \mathtt{D}(:\mathtt{d})\ \{\overline{\mathtt{M}}\}$ 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 $\Gamma \vdash \mathtt{T}\ \mathtt{e}$ 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 andT-SCAST for upwards cast, downwards cast, and "stupid" cast respectively.

T-INV is a form of contraction that permits the class invariant `c` of a class `C` to enrich the type of any variable of type `C`.

In T-FIELD, we postulate the existence of a receiver object `o` of the given static type ($\mathtt{T}_0$). $fields(\mathtt{T}_0, o)$ 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 $\mathtt{o}.\mathtt{f}_i = \mathtt{self}$.[1] 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)[2] 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.[3]

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 2.1 (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

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

THEOREM 2.2 (Progress). *If $\vdash \mathtt{T}\ \mathtt{e}$, then one of the following conditions holds:*

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

THEOREM 2.3 (Type Soundness). *If $\vdash \mathtt{T}\ \mathtt{e}$ and $\mathtt{e} \longrightarrow^* \mathtt{e}'$, then $\mathtt{e}'$ is either (1) a value $\mathtt{v}$ with $\vdash \mathtt{S}\ \mathtt{v}$ and $\vdash \mathtt{S} \sqsubseteq \mathtt{T}$, for some type $\mathtt{S}$, 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 2.4 (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 $\mathtt{S}$ 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 2.5 (Weakening). *If $\Gamma \vdash \mathtt{T}\ \mathtt{e}$, then $\Gamma, \mathtt{S}\ \mathtt{x} \vdash \mathtt{T}\ \mathtt{e}$.*

LEMMA 2.6 (Body type). *If $mtype(\mathtt{T}_0, \mathtt{m}) = \overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c} \to \mathtt{S}$, 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 $\overline{\mathtt{T}}\ \overline{\mathtt{x}}, \mathtt{U}_0\ \mathtt{this} \vdash \mathtt{V}\ \mathtt{e}$*

3. Applied constrained calculii

The following section presents examples using several different constraint systems.

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 `C` is $\mathtt{class}\ \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ldots$. Also, we abbreviate $\mathtt{C}(\overline{\mathtt{t}} : \mathtt{true})$ as $\mathtt{C}(\overline{\mathtt{t}})$. Finally, we will also have need to use the shorthand $\mathtt{C}_1(\overline{t}{:}\mathtt{c}_1)\&\ldots\mathtt{C}_k(\overline{\mathtt{t}}_k : \mathtt{c}_k)$ for the type $\mathtt{C}_1(:\overline{\mathtt{f}}_1 = \overline{\mathtt{t}}_1, \ldots, \overline{\mathtt{f}}_k = \overline{\mathtt{t}}_k, \mathtt{c}_1, \ldots, \mathtt{c}_k$ provided that the $\mathtt{C}_i$ form a subtype chain and the declared fields of $\mathtt{C}_i$ are $\mathtt{f}_i$.

Constraints naturally allow for partial specification (e.g. inequalities) or incomplete specification (no constraint on a variable) with the same simple syntax. In the example below, the type of `a` does not place any constraint on the second dimension of `a`, but this dimension can be used in other types (e.g., the return type).

```
class Matrix(int m, int n) {
  Matrix(m,a.n) mul(Matrix(:m=this.n) a) {...}
  ...
}
```

Constraints also naturally permit the expression of existential types:

[1] 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.

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

[3] 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.

```
class List(int length) {
List(:self.length < length) filter(Comparator k) { ...}
...
}
```

Here, the length of the list returned by the "filter" method is unknown, but is bound by the length of the original list.

3.1 X10

3.2 Self types and binary methods

Self types [**?**, **?**] can be implemented using a `klass` property on objects. The `klass` property represents the run-time class of the object. Self types can be used to solve the binary method problem [3].

In the example below, the `Set` interface has a "union" method whose argument must be of the same class as `this`. This enables the `IntSet` class's `union` method to access the `bits` field of its argument `s`.

```
interface Set(:Class klass) {
  Set(this.klass) union(Set(this.klass) s);
}
class IntSet(:Class klass) implements Set(klass) {
  long bits;

  IntSet(IntSet.class)() { property(IntSet.class); }

  IntSet(IntSet.class)(int(:0 <= self, self <= 63) i) {
    property(IntSet.class);
    bits = 1 << i; }

  Set(this.klass) union(Set(this.klass) s) {
    IntSet(this.klass) r = new IntSet(this.klass);
    r.bits = this.bits | s.bits;
    return r; }
}
```

The key to ensuring that this code type-checks is the T-CONSTR rule. With a constraint system $\mathcal{C}_{\mathsf{klass}}$ aware of the `klass` property, the rule T-VAR is used to subsume an expression of type `Set(this.class)` to type `IntSet(this.class)` when `this` is known to be an `IntSet`:

$$\frac{\texttt{IntSet this}, \texttt{Set(this.klass) s} \vdash \texttt{Set(this.klass) s} \quad \texttt{IntSet this}, \texttt{Set(this.klass) s} \vdash_{\mathcal{C}_{\mathsf{klass}}} \texttt{IntSet(this.klass) s}}{\texttt{IntSet this}, \texttt{Set(this.klass) s} \vdash \texttt{IntSet(this.klass) s}}$$

3.3 AVL trees and red–black trees

AVL trees can be modeled so that the data structure invariant is enforced statically.

```
class AVLList(int(:self >= 0) height) {...}
class Leaf(Object key) extends AVLList(0) {...}
class Node(Object key, AVLList l, AVLList r
         : int d=l.height-r.height; -1 <= d, d <= 1)
    extends AVLList(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 Tree(int blackHeight) {...}
class Leaf extends Tree(0) { int value; ...}
class Node(boolean isBlack,
           Tree(:this.isBlack || isBlack) l,
           Tree(:this.isBlack || isBlack,
                blackHeight=l.blackHeight) r)
    extends Tree(l.blackHeight+1) { int value; ... }
```

3.4 Bounds checks

Xi and Pfenning proposed using dependent types for eliminating array bounds checks [**?**].

In CFJ, an array of type `T[]` indexed by (signed) integers can be modeled as a class with the following signature:[4]

```
class 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);
}
```

Constraint system based on Presburger arithmetic:

```
a ::= b < b | b = b
b ::= t | n | b*b | b+b
```

Some code that iterates over an array (sugaring `get` and `set`):

```
double dot(double[] x, double[] y
         : x.length = y.length) {
  double r = 0.;
  for (int(:self >= 0, self < x.length)
       i = 0; i < x.length; i++) {
    r += x[i] * y[i];
  }
  return r;
}
```

3.5 Nullable types

A constraint system that supports disequalities can be used to enforce a non-null invariant on reference types. A non-null type `T` can be written simply as `T(:self != null)`.

3.6 A distributed binary tree

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:

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

3.7 Clocked types

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

For each arity n, we introduce a *Gentzen predicate* $\texttt{clock}(\overline{\texttt{t}})$. 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 $i \leq k$.

A programmer may

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

A programmer can require that a method may be invoked only if the

3.8 Places

3.9 k-dimensional regions

3.10 Point

3.11 Distribution

3.12 Arrays

Following ZPL [**?**], arrays in X10 are defined over sets of n-dimensional *index points* called *regions* [10]. For instance, the

[4] For this example, we assume the language supports generics.

```
1  class Owned(Owned owner) { }
2
3  class List(Owned valOwner
4           : owner contains valOwner)
5      extends Owned {
6    Owned(valOwner) head;
7    List(this, valOwner) tail;
8
9    List(:owner=o, valOwner=v, o contains v)
10       (Owned o, Owned v: o contains v) {
11     super(o);
12     property(v);
13   }
14
15   List(this, valOwner) expose() {
16     return tail;
17   }
18
19   ...
20 }
```

Figure 7. Ownership types

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 have the following syntax:

(atom)	a	::=	$\mathtt{r} \subseteq \mathtt{r}$
(region)	r	::=	$\mathtt{t} \mid [\mathtt{b}_1 : \mathtt{d}_1, \ldots, \mathtt{b}_k : \mathtt{d}_k] \mid \mathtt{r}\vert\mathtt{r} \mid \mathtt{r}\&\mathtt{r} \mid \mathtt{r}-\mathtt{r} \mid \mathtt{r}+\mathtt{p}$
(point)	p	::=	$\mathtt{t} \mid [\mathtt{b}_1, \ldots, \mathtt{b}_k]$
(integer)	b,d	::=	$\mathtt{t} \mid \mathtt{n}$

Constraints include subset constraints between regions. Regions used in constraints are either constraint terms, region constants, union (`|`), intersection (`&`), and set difference ($-$), or regions where each point is shifted by a point (`p`).

For example, the code in Figure 6 performs a successive over-relaxation [**?**] of an $n \times n$ matrix `G`. The type-checker establishes that the `region` property of the point `ij` (line 16) is `inner & [i,i:d1min,d1max]`, and that this region is a subset of `outer`, the region of the array `G`.

3.13 Clocks

Clock types

3.14 Capabilities

Capablities (from Radha and Vijay's paper on neighborhoods)

3.15 Ownership types

Figure 7 shows a fragment of `List` class, demonstrating how ownership types [4] can be encoded in CFJ. Each `Owned` object has an `owner` property. Objects also have properties used as owner parameters. The `List` class has a property `valOwner` that is instantiated with the owner of the values in the list, stored in the `head` field of each element. The `tail` of the list is owned by the list object itself.

To enforce the "owners as dominators" property, the owner of the values `valOwner` must be contained within the owner of the list itself; that is, `valOwner` must be `owner` or `valOwner`'s owner must be contained in `owner`. This is captured by the constraint `owner contains valOwner`.

The `expose` method incorrectly leaks the list's `tail` pointer. The constraint system catches this XXX.

3.16 Discussion

Control-flow. Tricky to encode. Need something like "pc" label [**?**].

Type state. Type state depends on the mutable state of the objects. Cannot do in this framework.

Dependent types are of use in annotations [16].

4. Implementation

The dependent type system is implemented in the X10 compiler [23], which is implemented as an extension of Java using the Polyglot compiler framework [15].

Polyglot implements a source-to-source base Java compiler that is extended to translate X10 to Java. For purposes of this paper, we ignore the additional statement and expression types introduced in X10 and treat the language as simply Java extended with constrained types.

Type-checking is implemented as two passes. The first pass performs type-checking using the base compiler implementation augmented with additional code for new statement and expression types introduced in X10. In this pass, expressions used in dependent types are type-checked using the non-dependent type system, however, no constraint solving is performed. The second pass generates and solves constraints via an ask–tell interface [22]. If constraints cannot be solved, an error is reported.

After constraint checking, the X10 code is translated to Java. The basic idea behind the translation is simple. 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, there is also a getter method in the 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, constraints are simply erased from the generated code. However, dependent types may be used in casts and `instanceof` expressions. These are translated to Java in straightforward manner 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⟧)
```

5. Conclusion and Future work

State-dependent constrained types

Use of dependent types for optimization

Constraints on control-flow

Type inference

We have presented a simple design for dependent types in Java-like languages. The design considerably enriches the space of (mostly) statically checkable types expressible in the language. This is particularly important for data-structurs such as lists and arrays. We have shown a simple translation scheme for dependent types into an underlying language with `assert` and `assume` statements. The assert and assume statements generated by this translation have the important property of state invariance. This enables a very simple notion of simplification for such programs. A general

```
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 outer = G.distribution.region;
5    region inner = outer & (outer + WEST)  & (outer - WEST)
6                         & (outer + NORTH) & (outer - NORTH);
7    region d0 = inner.project(0);
8    region d1 = inner.project(1);
9    if (d1.size() == 0) return;
10   int d1min = d1.min()[0];
11   int d1max = d1.max()[0];
12   for (point[off] : [1:iter*2]) {
13     int red_black = off % 2;
14     finish foreach (point[i] : d0) {
15       if (i % 2 == red_black) {
16         for (point ij : inner & [i:i,d1min:d1max]) {
17           G[ij] = omega * 0.25 * (G[ij-NORTH] + G[ij+NORTH]
18                                  + G[ij-WEST]  + G[ij+WEST])
19                 * (1. - omega) * G[ij];
20         }
21       }
22     }
23   }
24 }
```

Figure 6. Successive over-relaxation with regions

constraint propagator can simplify programs by using ask and tell operations on the underlying constraint system. Assert statements are removed if they are entailed by the conjunction of `assume`s on each path to the statement.

Our treatment is parametric in that the underlying constraint system can vary. Indeed the constraint system is not required to be complete; any incompleteness results merely in certain asserts being relegated to runtime. Some of these asserts may throw runtime exceptions if they are violated.

In future work we plan to investigate optimizations (such as array bounds check elimination) enabled by dependent types. We also plan to pursue much richer constraint systems, e.g., those necessary to deal with regions, cyclic and block-cyclic distributions etc.

Acknowledgments

Igor Peshansky, Lex Spoon, Vincent Cave.

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 and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, June 1993.

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

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

[4] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.

[5] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis*. PhD thesis, Cornell University, February 1990.

[6] 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.

[7] Manuel Fähndrich. *Bane: A library for scalable constraint-based program analysis*. PhD thesis, University of California Berkeley, 1999.

[8] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, 2002.

[9] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *European Symposium on Programming (ESOP)*, number 300 in LNCS, pages 94–114, March 1988.

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

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

[12] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(1):1811–1841, November 1994.

[13] 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.

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

[15] 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, pages 138–152, Warsaw, Poland, April 2003.

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

[17] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

[18] François Pottier. Simplifying subtyping constraints. In *Proceedings*

of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96), volume 31, pages 122–133, 1996.

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

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

[21] François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter 10, The Essence of ML Type Inference. MIT Press, 2004.

[22] V. Saraswat. *Concurrent Constraint Programming*. Doctoral Dissertation Award and Logic Programming. MIT Press, 1993.

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

[24] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

[25] 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.

[26] 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.

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

[28] 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.