Constrained Types for Object-Oriented Languages

Vijay Saraswat
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
vsaraswa@us.ibm.com

Nathaniel Nystrom
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
nystrom@us.ibm.com

Radha Jagadeesan
Depaul University
rjagadeesan@cs.depaul.edu

Jens Palsberg
University of California–Los Angeles
palsberg@cs.ucla.edu

Christian Grothoff
University of Denver
christian@grothoff.org

Abstract

We formalize the basic ideas of dependent-types for Java-like languages, in the context of FX10, a Featherweight version of X10.

1. Introduction

X10 is a modern statically typed object-oriented (OO) language designed for high productivity in the high performance computing (HPC) domain [22]. 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* [27].

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.

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* [5] 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.

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 `length`, 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 [13, 25], 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 3 fleshes out the syntactic and semantic details of the proposal. Formal semantics and a soundness theorem are presented in Section 4. Section 5 works through a number of examples using a variety of constraint systems. Sections 7 and 8 conclude the paper with a discussion of future work.

2. Related work

Constraint-based type systems:

[12]
[8]
[4]
[1]
[10]
[23]
[16]
[26]

Types of the form $T\backslash C$, where C is a set of subtyping constraints.

Not dependent.

[6]

Pottier [17, 19] presents a constraint-based type system for an ML-like language with subtyping.

HM(X) [24, 18, 20] 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.

- Pottier
- HM(X)
- Xi and Pfenning (POPL99)

  Dependent types in practical programming. Dependent ML. DML(C). Objects drawn from constraint domain C.

  Index objects must be pure. Singleton types int(n).

  ML_0^{Π}: Refinement of the ML type system: does not affect the operational semantics. Can erase to ML_0.
- Ada dependent types

  Ada has constrained array definitions. A constraint [**?**]. Not clear if their dependent. Are there other dependent types?
- singleton kinds (Chris Stone)
- Nested types, vObj, Scala, gbeta, J&

  Nested types: witness types, p vObj, Scala: p.type. J&: p.class gbeta: p.C
- Where clauses for F-bounded polymorphism (Theta and PolyJ)

  Bounded quantification: Cardelli and Wegner. Bound T with T' F-bounds: Canning, Cook, Hill, Olthoff, Mitchell. Bound T with F(T).

  Not dependent types.
- Hybrid type checking (Flanagan, POPL06)

  Refinement types. Types can be arbitrary predicates:

  z : Int — z ¿= 0 equivalent to the X10 type: Int(:self ¿= 0)

  Subtyping is undecidable. Type-checker can report "yes", "no", or "don't know". If the latter, dynamic checks inserted where subsumption occurs.

  Also Hybrid Types, Invariants and Refinements for Imperative Objects (Flanagan, Freund, Tomb, FOOL06).

  Constraints must be pure. Update is not pure. Calls must have pure actuals + receiver. What about alising?
- Soft typing with conditional types (Aiken, POPL94)

  Aiken, Wimmers, and T.K. Lakshman.
- Cayenne (Augustsson, ICFP98)

  Result type of function can depend on argument value. Any expression is a type. Haskell-like. Undecidable type checking.
- Cardelli, type checking dependent types and subtypes
- Esterel
- Findler, Felleisen, Contracts for higher-order functions (ICFP02)

  example: int[¿ 9]

  contracts are either simple predicates or function contracts. defined by (define/contract ...)

```
public value class List(int length : length >= 0) {
  Object head = null;
  List(length-1) tail = null;

  /** Returns the empty list. */
  public List(:length==0)() {
    property(0);
  }

  /** Returns a singleton list. */
  public List(:length==1)(Object head) {
    this(head, new List());
  }
  /** Returns the cons of head and tail. */
  public List(:length==tail.length+1)(Object head, final List tail) {
    property(tail.length+1);
    this.head = head;
    this.tail = tail;
  }
  public List(length+arg.length) append(final List arg) {
    return (length == 0)
          ? arg
          : new List(head, tail.append(arg));
  }
  public List(length) rev() {
    return rev(new List());
  }
  public List(length+acc.length) rev(final List acc) {
    return (length == 0)
          ? acc
          : tail.rev(new List(head, acc));
  }
  /** Return a list of compile-time unknown length, obtained by filtering
      this with f. */
  public List(:self.length <= this.length) filter(Predicate f) {
    if (length==0) return this;
    if (f.isTrue(head)) {
      List<Object> l = tail.filter(f);
      return new List(l+1)(head, l);
    } else {
      return tail.filter(f);
    }
  }
}
```

Figure 1. List example

enforced at run-time.

- Jif (final access paths in security labels)
- FX-90
- ESCJava [7], Spec#
- JSR 308, Javari
- Freeman, Pfenning, Refinement types for ML (PLDI91)
- Holt, Cordy, the Turing programming language
- Mandelbaum, Walker, Harper, effective thy of type refinements
- Ou, Tan, Mandelbaum, Walker, Dynamic typing with dependent types

  Separate dependent and simple parts of the program. Statically type the dependent parts. Dynamic checks when passing values into dependent part.
- Dependently typed data structurse (Xi)
- Dead code elimination through dependent types (Xi)

3. 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* [5] 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.

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.

Parametric 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 [11]), 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.

3.1 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.

3.1.1 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:

```
/** Returns the empty list. */
public List(:length==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. In a class has properties, every path through the constructor must contain exactly one property statement.

3.2 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.

3.3 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 parameters 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.*

3.4 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 parameteric 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.

3.5 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

4. Semantics

THEOREM 4.1 (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) $\mathtt{v} \quad ::= \quad \mathtt{new\ C}(\bar{\mathtt{v}})$

THEOREM 4.2 (Type Soundness). *If* $\vdash T$ *e and* $e \longrightarrow^* e'$*, then* e' *is either (1) a value with* $\vdash \mathtt{S}$ v *and* $\vdash \mathtt{S} \sqsubseteq \mathtt{T}$*, for some type* S*, or, (2) an expression containing a subexpression* $(\mathtt{T})\mathtt{new\ C}(\bar{\mathtt{v}})$ *where* $\not\vdash \mathtt{C} \sqsubseteq \mathtt{T}[\mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{self}]$.

5. Examples

Specific constraint systems used when developing specific examples.

5.1 Binary methods

The binary method problem [**?**].

Bruce and Cardelli and Castagna and Leavens and Pierce, On Binary Methods, TAPOS 1(3): 221–242, Fall 1995.

```
interface Set {
    Set(:self.class == this.class)
        union(Set(:self.class == this.class) s);
    boolean superSetOf(Set(:self.class == this.class) s);
}

class IntSet implements Set {
    long bits;

    Set(:self.class == this.class)
        union(Set(:self.class == this.class) s) {

        Set(:self.class == this.class) r = new IntSet();
        // need to be able to conclud that r : IntSet
        r.bits = this.bits | s.bits;
        return r;
    }

    boolean superSetOf(Set(:self.class == this.class) s) {
        return (s.bits & ~bits) == 0;
    }
}
```

5.2 Cayenne examples

5.3 DML examples

Red/black tree invariant.

Invariant: 1. all leaves black 2. for each node n, there are the same number of black nodes on every path from n to a leaf (the black height) 3. the immediate children of every red node are black

```
class Tree(boolean isBlack, int blackHeight, Tree left, Tree ri
  : ( left == null && right == null ||   // leaf or interior
      left != null && right != null
    )
    &&
    ( left == null || left.value < right.value )   // ordered
    &&
    ( left != null || isBlack )   // leaves are black
    &&
    ( left != null || blackHeight == 0 ) // leaves have height
    &&
    ( left == null || blackHeight == left.blackHeight + 1 )
    &&
    ( right == null || blackHeight == right.blackHeight + 1 )
    &&
    ( isBlack || ( left.isBlack && right.isBlack ) )
  )
{
    int value;

    // empty
    Tree(: isBlack && blackHeight == 0 && left == null && right
        () {
        property(isBlack, 0, null, null);
    }

    // black
    Tree(: isBlack && blackHeight == l.blackHeight+1 && left ==
        (Tree l, Tree r, int v : l.blackHeight == r.blackHeight
        property(true, l.blackHeight+1, l, r);
        this.value = v;
    }

    // red: use a dummy argument to overload
    Tree(: !isBlack && blackHeight == l.blackHeight && left ==
        (Tree l, Tree r, int v, Object _ : l.blackHeight == r.b
        property(false, l.blackHeight, l, r);
        this.value = v;
    }

    // need to show add, remove preserve the invariant
}
```

Bounds checks.

```
class Array {
    T[] a;
    T get(int(:0 <= self && self < a.length) i) { return a[i];
    void set(int(:0 <= self && self < a.length) i, T v) { a[i]
}
```

5.4 Nullable types

Nullable types (T(:self != null))

Syntax. The syntax for the language is specified as follows.
We assume 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 $[\bar{t}/\bar{x}]$ to a constraint c by $c[\bar{t}/\bar{x}]$.

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

Subtyping Judgements. 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 $\mathtt{C}(:\mathtt{d})\ \mathtt{x}$ with $\mathtt{d}[\mathtt{x}/\mathtt{self}]$.

$$\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T} \qquad \frac{\texttt{class}\ \mathtt{C}(:\mathtt{c})\ \texttt{extends}\ \mathtt{D}(:\mathtt{d})\{\ldots\}}{\vdash \mathtt{C} \sqsubseteq \mathtt{D}}$$

$$\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{D} \quad \sigma(\Gamma), \mathtt{c} \vdash_{\mathcal{C}} \mathtt{d}}{\Gamma \vdash \mathtt{C}(:\mathtt{c}) \sqsubseteq \mathtt{D}(:\mathtt{d})} \qquad \frac{\Gamma \vdash \mathtt{S} \sqsubseteq \mathtt{T} \quad \Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{U}}{\Gamma \vdash \mathtt{S} \sqsubseteq \mathtt{U}}$$

Type Judgements. Let `C` be a class declared as `class C(:c) extends D(:d){`$\bar{\mathtt{T}}\ \bar{\mathtt{f}}$`;K;`$\bar{\mathtt{M}}$`}`. We let $inv(\mathtt{C})$ stand for the conjunction `c,d` and (recursively) $inv(\mathtt{D})$. We bottom out with $inv(\mathtt{Object}) = \mathtt{true}$. We use $ctor(C)$ to name the constructor `K`.

$$\Gamma, \mathtt{T\ x} \vdash \mathtt{T\ x}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{C}(:\mathtt{c})\ \mathtt{x} \quad \mathtt{d} = inv(C)}{\Gamma \vdash \mathtt{C}(:\mathtt{c},\mathtt{d}[\mathtt{self}/\mathtt{this}])\ \mathtt{x}}\ \text{(T-Inv)} \qquad \frac{\Gamma \vdash \mathtt{C}(:\mathtt{c})\ \mathtt{x} \quad \sigma(\Gamma) \vdash_{\mathcal{C}} \mathtt{d}[\mathtt{x}/\mathtt{self}]}{\Gamma \vdash \mathtt{C}(:\mathtt{c},\mathtt{d})\ \mathtt{x}}\ \text{(T-Constr)}$$

$$\frac{\Gamma \vdash \mathtt{T}_0\ \mathtt{e} \quad fields(\mathtt{T}_0) = \bar{\mathtt{U}}(:\bar{\mathtt{d}})\bar{\mathtt{f}}}{\Gamma \vdash \mathtt{U}_i(:\mathtt{T}_0\ \mathtt{this};\mathtt{this.f} = \mathtt{self},\mathtt{d}_i)\,\mathtt{e.f}_i}\ \text{(T-Field)} \qquad \Gamma \vdash \mathtt{T}\ (\mathtt{T})\mathtt{e}\ \text{(T-Cast)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{T}\ \mathtt{e}_0, \bar{\mathtt{T}}\ \bar{\mathtt{e}} \\ mtype(\mathtt{T}_0, m) = \mathtt{c} \Rightarrow \bar{\mathtt{Z}}\ \bar{\mathtt{z}} \rightarrow \mathtt{S}(:\mathtt{d}) \\ \Gamma, \mathtt{T}_0\ \mathtt{this}, \bar{\mathtt{T}}\ \bar{\mathtt{z}} \vdash \mathtt{c}, \bar{\mathtt{T}} \sqsubseteq \bar{\mathtt{Z}}\end{array}}{\Gamma \vdash \mathtt{S}(:\mathtt{T}_0\ \mathtt{this};\bar{\mathtt{T}}\ \bar{\mathtt{z}};\mathtt{d})\ \mathtt{e}_0.\mathtt{m}(\bar{\mathtt{e}})}\ \text{(T-Invk)} \qquad \frac{\begin{array}{l}\Gamma \vdash \bar{\mathtt{T}}\ \bar{\mathtt{e}} \\ ctor(\mathtt{C}) = \mathtt{C}(:\mathtt{d})(\bar{\mathtt{Z}}\ \bar{\mathtt{f}}:\mathtt{c}) \\ \Gamma, \bar{\mathtt{T}}\ \bar{\mathtt{f}} \vdash \mathtt{c}, \bar{\mathtt{T}} \sqsubseteq \bar{\mathtt{Z}}\end{array}}{\Gamma \vdash \mathtt{C}(:\bar{\mathtt{T}}\ \bar{\mathtt{f}};\mathtt{self}.\bar{\mathtt{f}} = \bar{\mathtt{f}},\mathtt{d})\ \mathtt{new}\ \mathtt{C}(\bar{\mathtt{e}})}\ \text{(T-New)}$$

Method and Class Typing.

$$\frac{\bar{T}\ \bar{x}, C\ \mathtt{this}, c \vdash S\ e, S \sqsubseteq T}{\mathtt{T\ m}(\bar{\mathtt{T}}\,\bar{\mathtt{x}} : \mathtt{c})\{\mathtt{return\ e};\}\ \text{OK in C}} \qquad \frac{\begin{array}{l}K = C(\bar{S}\ \bar{g}, \bar{T}\ \bar{f} : c')\{\mathtt{super}(\bar{g});\mathtt{this}.\bar{f} = \bar{f};\} \\ fields(D) = \bar{S}\ \bar{g} \\ \bar{S}\ \bar{g}, \bar{T}\ \bar{f}, \mathtt{this}.\bar{g} = \bar{g}, \mathtt{this}.\bar{f} = \bar{f}, c' \vdash c, d \\ \bar{M}\ \text{OK in } C\end{array}}{\texttt{class}\ C(:c)\ \texttt{extends}\ D(:d)\ \{\bar{T}\,\bar{f};K\,\bar{M}\}\ \text{OK}}$$

Figure 2. Constrained FJ

5.5 Arrays

Array/region/distribution types (examples from X10) [9]

5.6 Places

Place types (examples from X10)

5.7 Clocks

Clock types

5.8 Capabilities

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

5.9 Ownership types

Ownership types [3]

5.10 Discussion

Dependent types are of use in annotations [15].

6. Implementation

The dependent type system is implemented in the X10 compiler [22], which is implemented as an extension of Java using the Polyglot compiler framework [14].

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

Computation:

$$\frac{fields(C) = \bar{C}\ \bar{f}}{(\texttt{new C}(\bar{\texttt{e}})).\texttt{f}_i \longrightarrow \texttt{e}_i}\ \text{(R-FIELD)}$$

$$\frac{mbody(\texttt{m}, \texttt{C}) = \bar{x}.\bar{e}_0}{(\texttt{new C}(\bar{\texttt{e}})).\texttt{m}(\bar{\texttt{d}}) \longrightarrow [\bar{d}/\bar{x}, \texttt{new}\ C(\bar{e})/\texttt{this}]\texttt{e}_0}\ \text{(R-INVK)}$$

$$\frac{\vdash C \sqsubseteq D[\texttt{new}\ C(\bar{\texttt{d}})/\texttt{self}]}{(\texttt{D})(\texttt{new C}(\bar{\texttt{d}})) \longrightarrow \texttt{new}\ C(\bar{\texttt{d}})}\ \text{(R-CAST)}$$

Congruence:

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0{}'}{\texttt{e}_0.\texttt{f} \longrightarrow \texttt{e}_0{}'.\texttt{f}}\ \text{(RC-FIELD)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0{}'}{\texttt{e.m}(\bar{\texttt{e}}) \longrightarrow \texttt{e}_0{}'.\texttt{m}(\bar{\texttt{e}})}\ \text{(RC-INVK-RECV)}$$

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

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

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0{}'}{(\texttt{C})\texttt{e}_0 \longrightarrow (\texttt{C})\texttt{e}_0{}'}\ \text{(RC-CAST)}$$

Figure 3. Reduction rules for Constrained FJ

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 [21]. 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.

For the most part, 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.

```
⟦e instanceof C(:c)⟧ =
  new Object() {
    boolean check(Object o) {
      if (o instanceof C) {
        C self = (C) o;
        return ⟦c⟧;
      }
      return false;
    }
  }.check(⟦e⟧)
```

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

7. Future work

State-dependent constrained types
Use of dependent types for optimization
Constraints on control-flow
Type inference

8. Conclusions

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 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.

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

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

[5] 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.

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

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

[8] 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.

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

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

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

[12] 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.

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

[14] 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.

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

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

[17] François Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31, pages 122–133, 1996.

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

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

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

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

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

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

[24] 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.

[25] 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.

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

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

A. An extended example

```
/**
   A distributed binary tree.
   @author Satish Chandra 4/6/2006
   @author vj
 */
//                              ____P0
//                             |     |
//                             |     |
//                           _P2  __P0
//                          |  | |   |
//                          |  | |   |
//                         P3 P2 P1 P0
//                          *  *  *  *
// Right child is always on the same place as its parent;
// left child is at a different place at the top few levels of the tree,
// but at the same place as its parent at the lower levels.

class Tree(localLeft: boolean,
           left: nullable Tree(& localLeft => loc=here),
           right: nullable Tree(& loc=here),
           next: nullable Tree) extends Object {
    def postOrder:Tree = {
        val result:Tree = this;
        if (right != null) {
            val result:Tree = right.postOrder();
            right.next = this;
            if (left != null) return left.postOrder(tt);
        } else if (left != null) return left.postOrder(tt);
        this
    }
    def postOrder(rest: Tree):Tree = {
        this.next = rest;
        postOrder
    }
    def sum:int = size + (right==null => 0 : right.sum()) + (left==null => 0 : left.sum)
}
value TreeMaker {
    // Create a binary tree on span places.
    def build(count:int, span:int): nullable Tree(& localLeft==(span/2==0)) = {
        if (count == 0) return null;
        {val ll:boolean = (span/2==0);
         new Tree(ll,  eval(ll => here : place.places(here.id+span/2)){build(count/2, span/2)},
           build(count/2, span/2),count)}
    }
}
```

A.1 Places

```
/**

 * This class implements the notion of places in X10. The maximum
 * number of places is determined by a configuration parameter
 * (MAX_PLACES). Each place is indexed by a nat, from 0 to MAX_PLACES;
 * thus there are MAX_PLACES+1 places. This ensures that there is
 * always at least 1 place, the 0'th place.

 * We use a dependent parameter to ensure that the compiler can track
 * indices for places.
 *
 * Note that place(i), for i <= MAX_PLACES, can now be used as a non-empty type.
 * Thus it is possible to run an async at another place, without using arays---
 * just use async(place(i)) {...} for an appropriate i.

 * @author Christoph von Praun
 * @author vj
 */

package x10.lang;

import x10.util.List;
import x10.util.Set;
```

```
public value class place (nat i : i <= MAX_PLACES){

    /** The number of places in this run of the system. Set on
     * initialization, through the command line/init parameters file.
     */
    config nat MAX_PLACES;

    // Create this array at the very beginning.
    private constant place value [] myPlaces = new place[MAX_PLACES+1] fun place (int i) {
return new place( i )(); };

    /** The last place in this program execution.
     */
    public static final place LAST_PLACE = myPlaces[MAX_PLACES];

    /** The first place in this program execution.
     */
    public static final place FIRST_PLACE = myPlaces[0];
    public static final Set<place> places = makeSet( MAX_PLACES );

    /** Returns the set of places from first place to last place.
     */
    public static Set<place> makeSet( nat lastPlace ) {
Set<place> result = new Set<place>();
for ( int i : 0 .. lastPlace ) {
    result.add( myPlaces[i] );
}
return result;
    }

    /**  Return the current place for this activity.
     */
    public static place here() {
return activity.currentActivity().place();
    }

    /** Returns the next place, using modular arithmetic. Thus the
     * next place for the last place is the first place.
     */
    public place(i+1 % MAX_PLACES) next()  { return next( 1 ); }

    /** Returns the previous place, using modular arithmetic. Thus the
     * previous place for the first place is the last place.
     */
    public place(i-1 % MAX_PLACES) prev()  { return next( -1 ); }

    /** Returns the k'th next place, using modular arithmetic. k may
     * be negative.
     */
    public place(i+k % MAX_PLACES) next( int k ) {
return places[ (i + k) % MAX_PLACES];
    }

    /**  Is this the first place?
     */
    public boolean isFirst() { return i==0; }

    /** Is this the last place?
     */
    public boolean isLast() { return i==MAX_PLACES; }
}
```

A.2 *k*-dimensional regions

```
package x10.lang;

/** A region represents a k-dimensional space of points. A region is a
 * dependent class, with the value parameter specifying the dimension
 * of the region.
 * @author vj
 * @date 12/24/2004
 */
```

```
public final value class region( int dimension : dimension >= 0 )  {

    /** Construct a 1-dimensional region, if low <= high. Otherwise
     * through a MalformedRegionException.
     */
    extern public region (: dimension==1) (int low, int high)
        throws MalformedRegionException;

    /** Construct a region, using the list of region(1)'s passed as
     * arguments to the constructor.
     */
    extern public region( List(dimension)<region(1)> regions );

    /** Throws IndexOutOfBoundException if i > dimension. Returns the
        region(1) associated with the i'th dimension of this otherwise.
     */
    extern public region(1) dimension( int i )
        throws IndexOutOfBoundException;


    /** Returns true iff the region contains every point between two
     * points in the region.
     */
    extern public boolean isConvex();

    /** Return the low bound for a 1-dimensional region.
     */
    extern public (:dimension=1) int low();

    /** Return the high bound for a 1-dimensional region.
     */
    extern public (:dimension=1) int high();

    /** Return the next element for a 1-dimensional region, if any.
     */
    extern public (:dimension=1) int next( int current )
        throws IndexOutOfBoundException;

    extern public region(dimension) union( region(dimension) r);
    extern public region(dimension) intersection( region(dimension) r);
    extern public region(dimension) difference( region(dimension) r);
    extern public region(dimension) convexHull();

    /**
       Returns true iff this is a superset of r.
     */
    extern public boolean contains( region(dimension) r);
    /**
       Returns true iff this is disjoint from r.
     */
    extern public boolean disjoint( region(dimension) r);

    /** Returns true iff the set of points in r and this are equal.
     */
    public boolean equal( region(dimension) r) {
        return this.contains(r) && r.contains(this);
    }

    // Static methods follow.

    public static region(2) upperTriangular(int size) {
        return upperTriangular(2)( size );
    }
    public static region(2) lowerTriangular(int size) {
        return lowerTriangular(2)( size );
    }
    public static region(2) banded(int size, int width) {
        return banded(2)( size );
    }

    /** Return an \code{upperTriangular} region for a dim-dimensional
     * space of size \code{size} in each dimension.
```

```
     */
    extern public static (int dim) region(dim) upperTriangular(int size);

    /** Return a lowerTriangular region for a dim-dimensional space of
     * size \code{size} in each dimension.
     */
    extern public static (int dim) region(dim) lowerTriangular(int size);

    /** Return a banded region of width {\code width} for a
     * dim-dimensional space of size {\code size} in each dimension.
     */
    extern public static (int dim) region(dim) banded(int size, int width);


}
```

A.3 Point

```
package x10.lang;

public final class point( region region ) {
    parameter int dimension = region.dimension;
    // an array of the given size.
    int[dimension] val;

    /** Create a point with the given values in each dimension.
     */
    public point( int[dimension] val ) {
        this.val = val;
    }

    /** Return the value of this point on the i'th dimension.
     */
    public int valAt( int i) throws IndexOutOfBoundException {
        if (i < 1 || i > dimension) throw new IndexOutOfBoundException();
        return val[i];
    }

    /** Return the next point in the given region on this given
     * dimension, if any.
     */
    public void inc( int i )
        throws IndexOutOfBoundException, MalformedRegionException {
        int val = valAt(i);
        val[i] = region.dimension(i).next( val );
    }

    /** Return true iff the point is on the upper boundary of the i'th
     * dimension.
     */
    public boolean onUpperBoundary(int i)
        throws IndexOutOfBoundException {
        int val = valAt(i);
        return val == region.dimension(i).high();
    }

    /** Return true iff the point is on the lower boundary of the i'th
     * dimension.
     */
    public boolean onLowerBoundary(int i)
        throws IndexOutOfBoundException {
        int val = valAt(i);
        return val == region.dimension(i).low();
    }
}
```

A.4 Distribution

```
package x10.lang;

/** A distribution is a mapping from a given region to a set of
 * places. It takes as parameter the region over which the mapping is
 * defined. The dimensionality of the distribution is the same as the
```

```
 * dimensionality of the underlying region.

   @author vj
   @date 12/24/2004
 */

public final value class distribution( region region ) {
    /** The parameter dimension may be used in constructing types derived
     * from the class distribution. For instance,
     * distribution(dimension=k) is the type of all k-dimensional
     * distributions.
     */
    parameter int dimension = region.dimension;

    /** places is the range of the distribution. Guranteed that if a
     * place P is in this set then for some point p in region,
     * this.valueAt(p)==P.
     */
    public final Set<place> places; // consider making this a parameter?

    /** Returns the place to which the point p in region is mapped.
     */
    extern public place valueAt(point(region) p);

    /** Returns the region mapped by this distribution to the place P.
        The value returned is a subset of this.region.
     */
    extern public region(dimension) restriction( place P );

    /** Returns the distribution obtained by range-restricting this to Ps.
        The region of the distribution returned is contained in this.region.
     */
    extern public distribution(:this.region.contains(region))
        restriction( Set<place> Ps );

    /** Returns a new distribution obtained by restricting this to the
     * domain region.intersection(R), where parameter R is a region
     * with the same dimension.
     */
    extern public (region(dimension) R) distribution(region.intersection(R))
        restriction();

    /** Returns the restriction of this to the domain region.difference(R),
        where parameter R is a region with the same dimension.
     */
    extern public (region(dimension) R) distribution(region.difference(R))
        difference();

    /** Takes as parameter a distribution D defined over a region
        disjoint from this. Returns a distribution defined over a
        region which is the union of this.region and D.region.
        This distribution must assume the value of D over D.region
        and this over this.region.

        @seealso distribution.asymmetricUnion.
     */
    extern public (distribution(:region.disjoint(this.region) &&
                                dimension=this.dimension) D)
        distribution(region.union(D.region)) union();

    /** Returns a distribution defined on region.union(R): it takes on
        this.valueAt(p) for all points p in region, and D.valueAt(p) for all
        points in R.difference(region).
     */
    extern public (region(dimension) R) distribution(region.union(R))
        asymmetricUnion( distribution(R) D);

    /** Return a distribution on region.setMinus(R) which takes on the
     * same value at each point in its domain as this. R is passed as
     * a parameter; this allows the type of the return value to be
     * parametric in R.
     */
```

```
    extern public (region(dimension) R) distribution(region.setMinus(R))
        setMinus();

    /** Return true iff the given distribution D, which must be over a
     * region of the same dimension as this, is defined over a subset
     * of this.region and agrees with it at each point.
     */
    extern public (region(dimension) r)
        boolean subDistribution( distribution(r) D);

    /** Returns true iff this and d map each point in their common
     * domain to the same place.
     */
    public boolean equal( distribution( region ) d ) {
        return this.subDistribution(region)(d)
            && d.subDistribution(region)(this);
    }

    /** Returns the unique 1-dimensional distribution U over the region 1..k,
     * (where k is the cardinality of Q) which maps the point [i] to the
     * i'th element in Q in canonical place-order.
     */
    extern public static distribution(:dimension=1) unique( Set<place> Q );

    /** Returns the constant distribution which maps every point in its
        region to the given place P.
    */
    extern public static (region R) distribution(R) constant( place P );

    /** Returns the block distribution over the given region, and over
     * place.MAX_PLACES places.
     */
    public static (region R) distribution(R) block() {
        return this.block(R)(place.places);
    }

    /** Returns the block distribution over the given region and the
     * given set of places. Chunks of the region are distributed over
     * s, in canonical order.
     */
    extern public static (region R) distribution(R) block( Set<place> s);


    /** Returns the cyclic distribution over the given region, and over
     * all places.
     */
    public static (region R) distribution(R) cyclic() {
        return this.cyclic(R)(place.places);
    }

    extern public static (region R) distribution(R) cyclic( Set<place> s);

    /** Returns the block-cyclic distribution over the given region, and over
     * place.MAX_PLACES places. Exception thrown if blockSize < 1.
     */
    extern public static (region R)
        distribution(R) blockCyclic( int blockSize)
        throws MalformedRegionException;

    /** Returns a distribution which assigns a random place in the
     * given set of places to each point in the region.
     */
    extern public static (region R) distribution(R) random();

    /** Returns a distribution which assigns some arbitrary place in
     * the given set of places to each point in the region. There are
     * no guarantees on this assignment, e.g. all points may be
     * assigned to the same place.
     */
    extern public static (region R) distribution(R) arbitrary();

}
```

A.5 Arrays

Finally we can now define arrays. An array is built over a distribution and a base type.

```
package x10.lang;

/** The class of all  multidimensional, distributed arrays in X10.

    <p> I dont yet know how to handle B@current base type for the
    array.

 * @author vj 12/24/2004
 */

public final value class array ( distribution dist )<B@P> {
    parameter int dimension = dist.dimension;
    parameter region(dimension) region = dist.region;

    /** Return an array initialized with the given function which
        maps each point in region to a value in B.
     */
    extern public array( Fun<point(region),B@P> init);

    /** Return the value of the array at the given point in the
     * region.
     */
    extern public B@P valueAt(point(region) p);

    /** Return the value obtained by reducing the given array with the
        function fun, which is assumed to be associative and
        commutative. unit should satisfy fun(unit,x)=x=fun(x,unit).
     */
    extern public B reduce(Fun<B@?,Fun<B@?,B@?>> fun, B@? unit);


    /** Return an array of B with the same distribution as this, by
        scanning this with the function fun, and unit unit.
     */
    extern public array(dist)<B> scan(Fun<B@?,Fun<B@?,B@?>> fun, B@? unit);

    /** Return an array of B@P defined on the intersection of the
        region underlying the array and the parameter region R.
     */
    extern public (region(dimension) R)
        array(dist.restriction(R)())<B@P>  restriction();

    /** Return an array of B@P defined on the intersection of
        the region underlying this and the parametric distribution.
     */
    public  (distribution(:dimension=this.dimension) D)
        array(dist.restriction(D.region)())<B@P> restriction();

    /** Take as parameter a distribution D of the same dimension as *
     * this, and defined over a disjoint region. Take as argument an *
     * array other over D. Return an array whose distribution is the
     * union of this and D and which takes on the value
     * this.atValue(p) for p in this.region and other.atValue(p) for p
     * in other.region.
     */
    extern public (distribution(:region.disjoint(this.region) &&
                                dimension=this.dimension) D)
        array(dist.union(D))<B@P> compose( array(D)<B@P> other);

    /** Return the array obtained by overlaying this array on top of
        other. The method takes as parameter a distribution D over the
        same dimension. It returns an array over the distribution
        dist.asymmetricUnion(D).
     */
    extern public (distribution(:dimension=this.dimension) D)
        array(dist.asymmetricUnion(D))<B@P> overlay( array(D)<B@P> other);

    extern public array<B> overlay(array<B> other);
```

```
  public value class List <Node> {
    public final nat n;   // is a parameter
    nullable Node node = null;
    nullable List<Node> rest = null;  // All assignments must check n = this.n-1.

    /** Returns the empty list. Defined only when the parameter n
        has the value 0. Invocation: new List(0)<Node>().
     */
    public List ( final nat n ) {
      assume n==0;
      this.n = n;
    }

    /** Returns a list of length 1 containing the given node.
        Invocation: new List(1)<Node>( node ).
     */
    public List ( final nat n, Node node ) {
      assume n==1;                          // From the constructor precondition.
      assert 0==0 : "DependentTypeError"; // For the constructor call.
      assert n>=1 : "DependentTypeError"; // For the this call.
      this(n, node, new List<Node>(0));
    }

    public List ( final nat n, Node node, List<Node> rest ) {
      assume n>=1;                                // From the constructor precondition
      assume rest.n==n-1 : "DependentTypeError"; // From the argument type.
      this.n = n;
      this.node = node;
      assert rest.n==n-1 : "DependentTypeError"; // For the field assignment.
      this.rest = rest;
    }

    public  List<Node> append( List<Node> arg ) {
      if (n == 0) {
          final List<Node> result = arg;
          assert n+arg.n == result.n : "DependentTypeError"; // For the return value
          return result;
      } else {
          assume rest.n == n-1;
          final List<Node> argval = rest.append(arg);
          assume argval.n == rest.n+arg.n;
          assert n+arg.n-1== argval.n : "DependentTypeError"; // For the constructor call.
          final List<Node> result = new List<Node>(n+arg.n, node, argval);
          assume result.n == n+arg.n;
          assert n+arg.n == result.n : "DependentTypeError"; // For the return value
          return result;
      }
    }
```

Figure 4. Translation of `List` (contd in Table 5).

```
    /** Assume given an array a over distribution dist, but with
     * basetype C@P. Assume given a function f: B@P -> C@P -> D@P.
     * Return an array with distribution dist over the type D@P
     * containing fun(this.atValue(p),a.atValue(p)) for each p in
     * dist.region.
     */
    extern public <C@P, D>
        array(dist)<D@P> lift(Fun<B@P, Fun<C@P, D@P>> fun, array(dist)<C@P> a);

    /**  Return an array of B with distribution d initialized
         with the value b at every point in d.
     */
    extern public static (distribution D) <B@P> array(D)<B@P> constant(B@? b);

}
```

EXAMPLE A.1. *The code for* `List` *translates as given in Table 4.*

```
  public  List<Node> rev() {
    final List<Node> arg = new List<Node>(0);
    assume arg.n = 0;                           // From the constructor call.
    final List<Node> result = rev( arg );
    assume result.n == n+arg.n;                  // From the method signature
    assert n == result.n : "DependentTypeError"; // For the return value.
    return result;
  }

  public  List(n+arg.n)<Node> rev( final List<Node> arg) {
    if (n==0) {
       assert n+arg.n == arg.n : "DependentTypeError"; // For the return value.
       return arg;
    } else {
      assert 1+arg.n-1=arg.n : "DependentTypeError"; // For the argument to the constructor
      final List<Node> arg2 = new List<Node>(1+arg.n,node, arg));
      assume arg2.n==1+arg.n;                     // From the constructor invocation
      final List<Node> restval = rest;            // Read from a mutable field of parametric type
      assume restval.n == n-1;                    // From the field read.
      final List(restval.n+arg2.n)<Node> result = restval.rev( arg2 );
      assume result.n=restval.n+arg2.n
      assert n+arg.n == result.n                  // For the return value
      return result;
  }

  /** Return a list of compile-time unknown length, obtained by filtering
      this with f. */
  public List<Node> filter(fun<Node, boolean> f) {
       if (n==0) return this;
       if (f(node)) {
         final List<Node> l = rest.filter(f);
         assert l.n+1-1==l.n : "DependentTypeError"; // For the constructor call
         return new List<Node>(l.n+1,node, l);
       } else {
         return rest.filter(f);
       }
  }

  /** Return a list of m numbers from o..m-1. */
  public static  List<nat> gen( final nat m ) {
       assert 0 <= m : "DependentTypeError";        // Precondition for method call.
       final List<nat> result = gen(0,m);
       assume result.n=m-0 : "DependentTypeError";  // From the method signature
       assert m == result.n : "DependentTypeError"; // For the return value
       return result;
  }

  /** Return a list of (m-i) elements, from i to m-1. */
  public static List<nat> gen(final nat i, final nat m) {
    assume i <= m;                                  // Method precondition.
    if (i==m) {
      assert m-i == 0 : "DependentTypeError";       // For the constructor call
      final List result = new List<nat>(m-i);
      assume result.n == 0;                         // From the constructor call.
      assert m-i == result.n : "DependentTypeError"; // For the return value.
      return result;
    } else {
      assert i+1 <= m : "DependentTypeError";       // For the method call.
      final List<nat> arg = gen(i+1,m);
      assume arg.n = m-(i+1);                       // From the method call.
      assert m-i-1 = arg.n;                         // For the constructor invocation.
      final List result = new List<nat>(m-i, i, arg);
      assume result.n = m-i;                        // From the constructor invocation.
      assert m-i == result.n : "DependentTypeError"; // For the return value
      return result;
    }
  }
```

Figure 5. Translation of `List` (continued).

B. Type-checking dependent classes

Each programming language—such as X10—will specify the base underlying classes (and the operations on them) which can occur as types in parameter lists. For instance, in the code for `List` above, the only type that appears in parameter lists is `int`, and the only operations on `int` are addition, subtraction, >=, ==, and the only constants are `0` and `1`. (This language falls within Presburger arithmetic, a decidable fragment of arithmetic.) The compiler must come equipped with a constraint solver (decision procedure) that can answer questions of the form: does one constraint entail another? Constraints are atomic formulas built up from these operations, using variables. For instance, the compiler must answer each one of:

```
n >= 2 |- n-1 >= 0
n >= 0, m >= 0 |- m+n >= 0
```

Ultimately, the only variables that will occur in constraints are those that correspond to `config` parameters and those that are defined by implicit parameter definitions. We need to establish that the verification of any class will generate only a finite number of constraints, hence only a finite constraint problem for the constraint solver.

Second, it should be possible for instances of user-defined classes (and operations on them) to occur as type parameters. For the compiler to check conditions involving such values, it is necessary that the underlying constraint solver be extended.

There are two general ways in which the constraint solver may be extended. Both require that the programmer single out some classes and methods on those classes as *pure*. (We shall think of constants as corresponding to zero-ary methods.) Only instances of pure classes and expressions involving pure methods on these instances are allowed in parameter expressions.

How shall constraints be generated for such pure methods? First, the programmer may explicitly supply with each pure method `T m(T1 x1, ..., Tn xn)` a constraint on `n+2` variables in the constraint system of the underlying solver that is entailed by `y = o.m(x1,..., xn)`. Whenever the compiler has to perform reasoning on an expression involving this method invocation, it uses the constraint supplied by the programmer. A second more ambitious possibility is that a symbolic evaluator of the language may be run on the body of the method to automatically generate the corresponding constraint.

Finally an additional possibility is that the constraint solver itself be made extensible. In this case, when a user writes a class which is intended to be used in specifying parameters, he also supplies an additional program which is used to extend the underlying constraint solver used by the compiler. This program adds more primitive constraints and knows how to perform reasoning using these constraints. This is how I expect we will initially implement the X10 language. As language designers and implementers we will provide constraint solvers for finite functions and `Herbrand` terms on top of arithmetic.