Adding dependent types to X10

Vijay Saraswat
IBM TJ Watson Research Center
PO Box 704, Yorktown Heights
NY 10598
(Draft Version 0.21)
(Please do not cite)
(Send comments to vsaraswa@us.ibnm.com.)

24 December 2004

Abstract

X10 is a new modern statically typed object-oriented (OO) language designed for high productivity in the high performance computing (HPC) domain. 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 2-dimensional regions, `int[5]`, the type of all arrays of `int` of length 5, and `int[region(2)]`, the type of all arrays over two dimensional regions. The underlying general idea is that of *dependent types* – types that are parametrized by *values* [2], just as generic types are types paramterized by other types.

In this paper we develop a general syntactic and semantic framework for user-defined dependent types in the context of modern class-based OO languages such as Java, C# and X10. The design supports the definition of dependent and generic interfaces, classes and methods. Run time casts to parametric types are permitted. As in staged languages, the design distinguishes between compile-time evaluation and run-time evaluation. Dependent 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; run-time casts involve arithmetic, not algebra (the values of all parameters are known). No extension of an underlying Virtual Machine is necessary, except as may be useful in improving efficiency (for example, eliminating array bounds checks).

We outline the design for a compiler which can use an extensible constraint-solver for type-checking. The compiler translates source programs into target programs without dependent classes or parametric types but with `assume` and `assert` statements. A general constraint-propagator that depends only on the operational semantics of the language and is independent of 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 may need to be performed at runtime.

1 Introduction

1.1 The basic idea

Our basic idea about the introduction of dependent types in class-based statically typed OO languages is as follows. Broadly, we follow the spirit of generic types, but use values instead of types.

We permit the definition of a class `C` to specify a *parameter list*, a list of typed parameters `(T1 x1, ..., Tk xk)` similar in syntactic structure to a method argument list. Each parameter in this list is treated as a `public final` instance field of the class. Thus for instance, we may specify a class `List` with a `nat length` parameter. We also permit the specification of a *where clause* 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.

Given such a class definition, what types can be constructed from it? If the class `C` has no parameters, the only type that can be constructed is the type `C`. However, if the class has parameters, a large set of possibilities opens up. Indeed, on very general principles one can argue that *any* boolean expression on the parameters should specify a type: the type of all instances of the class which satisfy the boolean expression. Thus, `List(:length > 3)` should be a permissible type, as should `List(:length =< 41)`.

Accordingly we define a *parametric type* to be simply the name of a class or interface followed by a where clause (a boolean expression on the parameters of that class). We call the name of the class or interface the *base class* of the parametric type, and the where clause the *condition* of the parametric type. The denotation (semantic interpretation) of such a type is the set of all instances of subtypes of the base class whose parameters satisfy the where clause. Clearly, for the denotation of a parametric type t to be non-empty it must be the case that the condition of the parametric type is consistent with the where clause (if any) of the base class of t. We shall require that the compiler ensure that the type of any variable declaration is non-empty.

For simplicity we shall permit the syntax `C(t1,..., tk)` for the type `C (:x1 = t1 && ...&& xk = tk)` (assuming that the parameter list for `C` specifies the `k` parameters `x1,..., xk`, and each each term `ti` is of the correct type). Thus `List(n)` is the type of all lists whose length is `n`. We shall continue to use `C` as a type as well; it corresponds to the (vacuously) parametric type `C(:true)`, i.e. the type of all instances of `C`.

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 is clearly reasonable (i.e. satisfies Liskov's Substitution Principle). It implies that $C(: e)$ is a subtype of $C(: e')$ if e implies e'. Specifically, $C(: e)$ is a subtype of C, for all e. $C(: e)$ is empty exactly when e conjoined with the class invariant of C is inconsistent.

Preconditions on parameters. We will permit constructors and methods to specify preconditions (as where clauses) on parameters. For an invocation of a method (or constructor) to be type-correct it must be the case that the associated where clause is statically known to be satisfied. The return type of a method may contain expressions involving the parameters of arguments to the method. However, we will require tha any argument used in this way must be declared `final`. For instance:

```
public List(length+arg.length) append( final List arg) {...}
```

will be a valid method declaration. It says that `append` method invoked on a list with length `length` and with an argument `arg` will return a list whose length is `length+arg.length`.

Dependent methods. We will permit *dependent methods* (similar to generic methods): a method may specify a list of parameters in the same way that a class may. The parameters may be used to construct dependent types for the arguments and return value of the method.

Dependent, Generic types. We will permit a type to be simultaneously a dependent and a generic type. For instance, we will permit the type:

```
List(3)<Point>
```

This is the type of all lists with `3` elements, each of which is a `Point`.

Static parameters. We will permit *static parameters*. A static parameter is a `parameter constant` field of a class, i.e. a static final field explicitly marked as a parameter. Some fields may be marked as `config` fields: `config` fields are implicitly `parameter constant` and their values are assigned from the invocation environment for the program (e.g. command line or a configuration file).

All expressions appearing in parametric types must reference only constants or instance or static parameters visible at that type or variables declared `final`. Further, they must only involve methods that have been specially marked as parametric. This facilitates compile-time type-checking, as discussed below.

Properties of the design. A few points are worth emphasizing. By insisting that the condition of a parametric type reference only parameters of the base type, we ensure that whether an object belongs to a parametric type or not does not depend on the mutable state of the object, since parameters are `final` instance fields of the object. That is, the status of the predication "this object belongs to this parametric type" does not change over the lifetime of the object. Second, by insisting that each parameter be a *field* of the object, we are enforcing the reasonable property that 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. Recall further that X10 requires the immutable state of an object (the "value kernel") to be transmitted with the reference to an object; thus a remote place may evaluate casts of objects to parametric types using only information available with the reference to the object.

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

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 where clause for a class explicitly captures conditions on the parameters of the class that must be satisfied by any instance of the class. Thus the X10 compiler's static check for non-emptiness of the type of any variable captures these invariant violations at compile-time.

Rest of this paper In the next section I flesh out the syntactic details of the current proposal and consider a small example, `List(n)`. After that I present the code for `place`, `region`, `point`, `distribution` and `arrays` in X10. Most of the methods are `extern` methods (defined natively); however their signature shows how considerable information about the structure of the datatypes and the operations on them can be expressed through the type system.

2 Dependent Types

A dependent type is a type that takes values as arguments; such a value is called a *parameter*. Thus a dependent type is just like a generic type except that the arguments to the type are values, not types. Syntactically, these values are enclosed in parentheses, rather than in angle brackets. For instance:

```
List(10)<Cell> // The type of all lists of Cells of length 10.
```

The primary mechanism for the introduction of dependent types in X10 is the *dependent class declaration.*

2.1 Dependent class definitions

Dependent classes are specified by providing a parameter list (similar in structure to a method definition) right after the name of the class. This list is said to define the *explicit parameters* of the class. Each parameter introduces a `final` instance field of the same name and type in the class; hence parameter names must be distinct from field names. The list of parameters may be followed by a ":" (read as "where") and a boolean expression involving the explicit parameters (called the *where clause*). An instance of a class can only be created for actual values for the explicit parameter which can statically be known to satisfy the where clause.

Example 2.1 (List) *Consider the class* `List`*:*

```
public value class List( nat n ) <Node> {
  nullable Node node = null;
  nullable List(n-1)<Node> rest = null;

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

  /** Returns a list of length 1 containing the given node.
      Invocation: new List(1)<Node>( node ).
   */
  public List (:n=1) (Node node) {
    this(node, new List(0)<Node>());
  }
  public List (:n >= 1) {Node node, List(n-1)<Node> rest) {
      this.node = node;
      this.rest = rest;
  }
  public  List(n+arg.n)<Node> append(final List<Node> arg) {
    return
        (n == 0) ? arg : new List(n+arg.n)<Node>(node, rest.append(arg));
  }
  public  List(n)<Node> rev() {
    return rev(new List(0)<Node>());
  }
  public  List(n+arg.n)<Node> rev( final List<Node> arg) {
    return
      (n == 0)
      ? arg
      : rest.rev(new List(1+arg.m)<Node>( node, arg));
  }
  /** 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)) {
         List<Node> l = rest.filter(f);
         return new List(l+1)<Node>(node, l);
       } else {
         return rest.filter(f);
       }
  }
```

```
  /** Return a list of m numbers from o..m-1. */
  public static (nat m) List(m)<nat> gen() {
       return gen(0,m)();
  }
  /** Return a list of (m-i) elements, from i to m-1. */
  public static (nat i, nat m: i =< m) List(m-i)<nat> gen() {
    return (i == m)? new List(m-i)<nat>() : new List(m-i)<nat>(i, gen(i+1,m));
  }
}
```

Intuitively, this definition states that a `List` *object can only be created when supplied with a* `nat` *parameter* `n` *(intended to represent the length of the list). The class has two fields intended to hold the data item in the head of the list and the rest of the list. The nullary constructor is defined only if the value of the parameter is* `0`*. The unary constructor is defined only if the value of the parameter is* `1`*. Similarly for the binary constructor.*

Functions that append one list to another or that reverse a list can be defined quite naturally. In both of these 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 size cannot be known statically (it depends on properties of the argument function* `f` *which are not captured statically).*

Finally, the example illustrates a dependent method. This example is discussed more below.

Implicit parameters. A dependent class may also specify additional parameters in the body of the class. Such parameters are called *implicit* parameters. Syntactically, an implicit parameter is specified using instance field declaration syntax and prefixing the field declaration with the keyword `parameter`. The declaration of parameters must include an initializer which may depend only on parameters defined in the parameter list for the class or on previously defined parameters in the class body.

In what follows by the *parameters* of a (dependent) class we will mean either the explicit parameters enumerated in the parameter list, or the implicit parameters defined using a `parameter` declaration in the body of the class. Parameters may be used in an expression specifying the value of some other parameter (in a constructor or method call).

Example 2.2 *The class* `List` *may be extended to define an implicit* `boolean` *parameter* `isEven` *which records whether the length of the list is even.*

```
public value class List(int n : n >= 0) <Node> {
  parameter boolean isEven = (n % 2 = 0);
  ...
}
```

The parameters of a class may be used in any type expression in the body of the class at which they are visible. Parameters have the status of final instance fields, hence they are not visible in static fields, initializers or methods, or when they are overridden.

Creating instances of a dependent class. An instance of a dependent class is created using a `new` operation and passing an actual list of parameters (of the right type) after the class name and before the generic type parameters (if any). For instance:

```
new List(0)<Node>();
new String(10) (``abcdefghij'');
```

Dependent constructors. Sometimes a constructor for class `C` can only be invoked if the parameters of the class satisfy a certain condition. For instance, a nullary constructor for `List` is only defined when the parameter `n` has the value `0`.

Such conditions 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 parameters of the class.

Consider the definition of the nullary constructor for instance:

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

Any attempt to invoke this constructor with a parameter that cannot be determined at compile-time to equal `0` will cause a compiler error.

Dependent methods. Instance method declarations may specify parameters after the qualifiers and before the return type of the method, through a parameter list (called the *parameter list* of the method[1]). The parameter list may be followed by a where expression on the parameters of the method and/or the parameters of the class. (Recall that a where expression is a boolean expression following a "`:`".)

The parameter list of the method is visible in the declaration of the method and the body of the method. Each parameter is treated as a `final` local variable in the body of the method. The types of arguments to the method, and the return type of the method, may be built from type expressions referencing these parameters.

A method invocation must specify the parameters in an actual parameter list after the name of the method and before the arguments to the method. For each such method invocation, the compiler must determine the corresponding method definition and statically check that its where clause (if any) is satisfied by the invocation.

MetaNote Need to determine rules for overriding for generic methods and compare with the rule above.

Dependent methods are useful in particular when a parameter in the type of the result is not functionally determined by parameters of arguments. For instance, suppose we want to write a method to return the list of all numbers upto n. This method cannot take n as an argument because then it would not be able to assert that the return type of the method is `List(n)` (all variables that can be referenced in the return type of a method must be parameters visible to the method). Thus instead of writing

```
public static List<nat> gen(nat m) {
   // returns the list of numbers from 0 to m.
}
```

one may write:

```
public static (nat m) List(m)<nat> gen() {
   // returns the list of numbers from 0 to m.
}
```

Such a method may be invoked as:

```
List(32)<nat> l = C.gen(32)();
```

Extending dependent classes. *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.*

[1] We shall use the term "arguments" or "argument list" for the usual arguments passed to a method.

2.2 Types defined by a Dependent Class definition

A dependent class definition introduces many types.

Let `C` be a class declared with explicit parameters `x1,..., xk` of type `T1,..., Tk`. A parametric type is an expression of the form `C (:e)` where `e` is a boolean expression in the parameters of the class. We permit the syntactic abbreviation `C(t1,..., tk)`, for values `t1,..., tk` of type `T1,..., Tk` respectively for the type `C(:x1=t1 && x2=t2...&& xk=tk)`. If `C` is a generic class with type parameters `<T1,..., Tm>`, then any type expression based on `C` must also specify the generic type parameters.

An object is an instance of a parametric type if its an instance of the base type, and the values of its parameters satisfy the boolean condition.

Example 2.3 *The expression* `List(3)<Node>` *is a type – the type of all lists of* `Node` *whose* `n` *field contains the value* `3`*. By design of the* `List` *class, such a list has length* `n`*.*

The expression `List (: isEven) <Node>` *is a type – the type of all lists of* `Node` *of even length.*

2.3 Dependent interfaces

3 Understanding Dependent Types via translation.

One may understand dependent classes and parametric types via a simple, compositional source-to-source translation into a language without these features, but with `assert` and `assume` statements (as in modern Java like languages). A program simplifier may flow `assume` assertions around the program, treating them as constraints. It may simplify or eliminate `assert` statements using logical reasoning over constraints. If all `assert` statements are eliminated by the simplifier, then the program is said to be *statically correct*. Otherwise some residual checks are left for runtime and may generate runtime errors.

Thus compile-time type-checking is seen as a way of performing some runtime calculations early and ensuring that certain exceptions are not thrown at runtime.

3.1 The target language

The target language for the translation is X10 without dependent or parametric types, but with `assert` and `assume` statements, which we now define.

For `e` a boolean expression the statement:

```
assume e;
```

asserts that `e` will be true whenever control reaches that point in the code. This assertion is categorical – it does not depend on any property of the code other than what is expressed through the type system and hence checked statically. Thus it is a mere comment – it does not translate into any runtime code. A programmer may not add `assume` statements; only the compiler may add such statements e.g. when translating an enhanced type system into a base type system.

On the contrary the statement:

```
assert e : s;
```

(where `e` is a boolean expression and `s` a string) is an assertion that says `e` *should* be true at runtime whenever control reaches that point. In case it is not, a runtime error should be thrown with the given string as argument. Thus it translates into the code:

```
if (! e) throw new RuntimeError( s );
```

We permit the user to add arbitrary `assert` statements to the program.

State invariant predications. Certain kinds of predications play a special role in `assert` and `assume` statements. They permit a very direct correspondence between program execution and constraint-solving and are central to the idea that a uniform constraint-propagator may eliminate `assert` statements based on the logical content of `assume` statements.

We say that a predication (`boolean` valued expression) is *state invariant* if its value is independent of any mutations to any variable (or object referenced by a variable) involved in the expression. Specifically, any variable occurring in the predication must be a `final` variable, any field accesses must be to `final` fields, and any method occuring in the predication must be *state invariant* (see below). For instance, the predication `list.n+1==3` for a `final` variable `list` of type `List` is state-invariant; it depends only on the `final` field `n` of the `final` variable `list`, the constant `3` and the operation `+` which is (intuitively) state invariant.

We permit user-defined methods to occur in state invariant predications as long as they satisfy certain properties. Let m be a method on a class C, with k arguments. We say that it is *state invariant* is it satisfies the property that the set of tuples of objects on which it succeeds is closed under equivalence of objects. Two objects (of the same class) are said to be equivalent if their immutable state is identical. Thus a state invariant method satisfies the property that if on being invoked with arguments $a_1, \ldots, a_k$ it returns an object a, then on being invoked with $a'_1, \ldots, a'_k$ where each a_i is equivalent to a_i it will return a value a' which is equivalent to a.

Further such a method m must be associated with a *logical invariant*, a constraint (in the underlying constraint language) of arity $k+1$, which is a finite representation of the (infinite) *extension* of the method. The extension of a method m which takes k arguments is the set of all tuples $\langle t_1, \ldots, t_k, t\rangle$ which satisfy the property that when m is invoked on $t_1, \ldots, t_k$ it returns t.

Thus the constraint is a symbolic representation of the method and can be used for compile-time symbolic reasoning about the method. At runtime we can use the code of m to evaluate a call to m, since all the values of the method arguments are known. At compile-time, the actual values to a method call are not known. Instead we must reason about the execution of the body of a method assuming simply that the arguments are some arbitrary but unknown values (i.e. assuming that they are *logical variables*). Thus we must use the logical invariant associated with the method.

For instance each operation on `nat` (e.g. "`+`","`*`", "`%`" is state invariant and may be associated with a logical invariant. For instance the "`+`" operation may be associated with the constraint `x+y=z` in an arithmetic constraint solver.

Simplification of programs with state-invariant `assume` and `asserts`. As we shall see below, the translation of a program with dependent and parametric types introduces only those `assert` and `assume` statements whose predications are state invariant. (This is easy to see, dependent classes are designed to express only those properties of objects which are state invariant.)

State-invariant `assume` and `asserts` are quite easy to reason with, since the value of the expressions occurring in these assertions does not change with program execution. Thus the compiler may transform `assume e` to a *tell* of the constraint c_e obtained by simply translating e (treating each program variable as a logical variable, and replacing each method call in e with the corresponding logical invariant). An assert statement s of the form `assert e` can be eliminated if the conjunction of tell constraints on each path leading into it entails c_e, the constraint obtained from e. (If a path is guarded by a conditional, then the condition may be assumed on the positive path, and the negation on the negative path.) The ask and tell constraint languages of [1] may be used fruitfully in this context.

3.2 The translation

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 parameters of the dependent class are translated into `public final` (instance) fields of the target class.

Each constructor of the target class takes a `final` additional argument for each explicit parameter, and initializes the correspoding field to that value. Each constructor `assume`s the constructor's where clause (if any) at the beginning of the constructor, and `assert`s the class's where condition at the end of the constructor. Within the body of each constructor a call to a constructor for the superclass is treated like a method call: an `assert` is added for the precondition of the super constructor, and for the arguments of the call to the superconstructor. After the call, an `assume` is added for the super-class's where clause applied to `this`.

A dependent method of the source class with k parameters is translated into a method in the target class of the same name with k additional `final` arguments.[2] The body of the target method `assume`s the arguments satisfy the source method's where clause. It may also `assume` the class's where clause on `this` at the begining of the method. After that it executes the translated body of the source method.

Now we consider the translation of parametric types in the body of methods. For simplicity let us assume that the body of the method is translated into a "single assignment" form with nested method calls "flattened". Every argument to a method invocation at a parametric type is a `final` local variable. If a method has a parametric return type then every `return` statement is of the form `return v` where `v` is a `final` local variable. Similarly the value read from every mutable variable of parametric type (e.g. local variable, field) is read into a `final` variable of the same type (i.e. reads of mutable variables of parametric type occur only on the RHS of an assignment to a `final` local variable). The value written into every mutable variable of parametric type is read from a `final` variable of the same type.

Every statement in X10 can be transformed into an equivalent such statement by introducing enough new `final` local variables. For instance the method:

```
public  List(n+arg.n)<Node> rev( final List<Node> arg) {
  return
    (n == 0)
    ? arg
    : rest.rev(new List(1+arg.n)<Node>( node, arg));
}
```

is translated to:

```
public  List(n+arg.n)<Node> rev( final List<Node> arg) {
  if (n==0) {
     return arg;
  } else {
    final List(1+arg.n)<Node> arg2 = new List(1+arg.n)<Node>(node, arg);
    final List(n-1)<Node> restval = rest;              // Read from a mutable field of parametric type
    final List(restval.n+arg2.n)<Node> result = restval.rev( arg2 );
    return result;
}
```

Let `v` be a local variable or field declared at a parametric type `C(:b)`. The translation will declare `v` at type `C`. Every write to `v` must be of the form `v=r` where `r` is a final variable, per the reasoning above. The write must be preceded by the statement `assert r.b :  "DependentTypeError";` where `r.b` stands for the clause `b` with each parameter `p` in `b` replaced by `r.p`. Every read of `v` must be of the form `r=v` where `r` is a `final` variable; such a read is succeeded by the statement `assume r.e`.

Formal arguments to methods of a parametric type are treated similarly.

Similarly a parametric type occurring as the return type of a method is replaced in the translation by the base type. Each expression `return e;` in the body of the method is translated as a write of `e` to a local variable declared at the parametric type.

[2]For simplicity of presentation we are assuming that the class does not already have a method with the same name and k extra arguments.

Every method invocation is considered as a write to the formal arguments of the method, and hence preceded by `assert`s if the type of a formal argument is parametric. It is also preceded by an `assert` of the method precondition, if any. Every method invocation is succeded by an `assume` of the return type of the method applied to the return value.

For instance the code above is translated to:

```
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<Node> result = restval.rev( arg2 );
    assume result.n=restval.n+arg2.n
    assert n+arg.n == result.n                    // For the return value
    return result;
}
```

Simplifcation The statement in the body of the method now generates the following constraint operations. Here we assume that `arg` and `n` `restval`, `result`, `arg2` are free `nat` variables.

There is a single path to the first ask, guarded by `n=0`. Thus the constraint problem to be solved is:

```
n=0,  (n+arg.n = arg.n => Remove1)
```

The underlying constraint system can solve the problem:

```
n=0 |-  (n+arg.n = arg.n)
```

thereby establishing `Remove1`. Hence the first `assert` can be removed.
There is a single path to the second and third asks, guarded by `n!=0`. The constraint problem generated is:

```
(n!= 0),  (1+arg.n-1 =arg.n) => Remove2
1+arg.n-1 = arg.n,
arg2.n =1+arg.n,
restval.n = n-1,
result.n = restval.n+arg2.n,
(n+arg.n = result.n) => Remove3
```

The underlying constraint solver can establish

```
|- 1+arg.n-1=arg.n
```

without using any constraint at all (this is vacuously true). Hence `Remove2` is established. It can also establish:

```
arg2.n =1+arg.n, restval.n = n-1, result.n = restval.n+arg2.n |- n+arg.n=result.n
```

through simple substitution. Hence `Remove3` is established.

For the `List` program, all asserts can be removed. The resulting program is given in Table 1-2.

```
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.

  public List ( final nat n ) {
    assume n==0;
    this.n = n;
  }
  public List ( final nat n, Node node ) {
    assume n==1;                          // From the constructor precondition.
    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;
    this.n = n;
    this.node = node;
    this.rest = rest;
  }

  public  List<Node> append( final List<Node> arg ) {
    if (n == 0) {
        return arg;
    } else {
        final List<Node> restval = rest;
        assume restval.n == n-1;
        final List<Node> argval = restval.append(arg);
        assume argval.n == restval.n+arg.n;
        final List<Node> result = new List<Node>(n+arg.n, node, argval);
        assume result.n == n+arg.n;
        return result;
    }
  }
```

Table 1: Translation of `List` (contd in Table 2).

```
  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
    return result;
  }

  public  List<Node> rev( final List<Node> arg) {
    if (n==0) {
       return arg;
    } else {
      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<Node> result = restval.rev( arg2 );
      assume result.n=restval.n+arg2.n
      return result;
  }

  public List<Node> filter(fun<Node, boolean> f) {
       if (n==0) return this;
       if (f(node)) {
         final List<Node> l = rest.filter(f);
         return new List<Node>(l.n+1,node, l);
       } else {
         return rest.filter(f);
       }
  }


  public static  List<nat> gen( final nat m ) {
       final List<nat> result = gen(0,m);
       assume result.n=m-0 : "DependentTypeError";  // From the method signature
       return result;
  }

  public static List<nat> gen(final nat i, final nat m) {
    assume i =< m;                                  // Method precondition.
    if (i==m) {
      final List result = new List<nat>(m-i);
      assume result.n == m-i;                       // From the constructor call.
      return result;
    } else {
      final List<nat> arg = gen(i+1,m);
      assume arg.n = m-(i+1);                       // From the method call.
      final List result = new List<nat>(m-i, i, arg);
      assume result.n = m-i;                        // From the constructor invocation.
      return result;
  }
}
```

Table 2: Translation of `List` (contd.

4 Conclusion

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 he 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.

References

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

[2] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.

A More extended examples

Below I discuss the problem of specifying the operations on places, regions, distributions and arrays in X10. I specify the signatures of these classes, making extensive use of dependent types.

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);

    /** 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);

}
```