An annotation and compiler plugin system for X10

A High-Level Design Document

Nathaniel Nystrom and Vijay Saraswat
IBM T. J. Watson Research Center
{nystrom,vsaraswa}@us.ibm.com

February 21, 2007

Abstract

We propose an annotation system and pluggable type system for X10. The system allows X10 programmers to annotate syntactic constructs in a program. Annotations are examined by compiler plugins, which may implement additional static checking and program transformations guided by the annotations. This paper describes the syntactic and semantic extensions to X10 for supporting annotations and the plugin interface for the X10 compiler.

1 Introduction

Annotations and compiler plugins are a mechanism for allowing programmers to extend the semantics of a programming language. These extensions are useful for many reasons:

- **Additional semantic checking.** Annotations can be used to refine the type information in the program. Plugins can check that the program conforms to the refined types. Plugins can check that coding conventions are followed.
- **Optimization.** Application-specific optimizations can be enabled by compiler plugins. Annotations can guide the transformations performed by the plugin.
- **Documentation.** Annotations provide a way to document the source code. Compiler plugins and tools can check that the documentation is accurate.
- **Language evolution.** Some aspects of programming language semantics (e.g., the memory model) tie the language to particular architectures or particular implementations. The semantics may not be suitable in some environments, or these features may be difficult to define precisely, leading to errors in the language definition. Annotations provide a way to extend or modify the semantics of a programming language without redefining the language itself and without breaking existing implementations. The language can evolve and adapt to new environments.

This report describes an annotation and compiler plugin system for X10 [14, 3]. The system allows X10 programmers to add annotations to syntactic constructs in a program and have those annotations examined by compiler plugins, which may implement additional static checking and program transformations guided by the annotations.

The system has a number of goals:

- **Orthogonality.** All types, expressions, statements, and declarations should be annotatable.

- **Expressiveness.** Annotations should support a rich syntax.
- **Extensibility.** Annotations should be extensible.
- **Type-safety.** Annotations should be statically type-checked in the base X10 language with additional processing provided by compiler plugins.
- **Configurability.** Compiler plugins should be configurable to be either *optional* or *mandatory* [2]. An annotated program should still compile when optional plugins are disabled.
- **Transformation support.** Compiler plugins should be able both to statically check code and transform code, including inserting run-time checks.

The syntactic and semantic extensions to X10 for supporting annotations are described. We describe how the X10 type system can be refactored so that dependent type constraints are implemented in compiler plugins, enabling type constraints to be extensible. The plugin interface and several example annotations are presented.

Although X10 is based on the Java programming language, the design makes no attempt to be backward compatible with Java 5 annotations [8] or the proposed JSR 308 annotations [9]. The design does not support Java's run-time retention framework for annotations, although a plugin may implement this functionality.

1.1 Architecture

Annotations are X10 interface types that decorate the abstract syntax tree (AST) of a program. The architecture of the plugin system is shown in Figure 1. The base X10 type-checker checks that annotations are legal interface types. Except for this type-checking of the annotations themselves, all semantic checking by the base X10 compiler (e.g., subtyping checks, cast checks, member lookups, binary promotion) ignores annotations. Annotations are not propagated through the AST during type checking (e.g., from a node for `x` to the node for `x+1`); this is left to the plugin.

After the base X10 semantic checking is completed, compiler plugins are loaded and run. Plugins may perform any number of compiler passes to implement additional semantic checking and code transformations, including transformations using the abstract syntax of the annotations themselves. Plugins should output valid X10 abstract syntax trees. After the plugins are executed, target code is generated from the X10 AST via one or more compiler passes.

1.2 Outline

Section 2 introduces the X10 annotation syntax and describes how annotations are type-checked. Annotation declarations are introduced in Section 3. Section 4 describes the built-in annotations in X10, and Section 5 presents some examples annotations. The compiler plugin infrastructure is described in Section 6. Related work is discussed in Section 7. The paper concludes in Section 8.

2 Annotations

Annotations are interface types that decorate the abstract syntax tree of an X10 program. The X10 type-checker ensures that an annotation is a legal interface type. In X10, interfaces may declare both methods and properties. Therefore, like any interface type, an annotation may instantiate one or more of its interface's properties. Unlike with Java annotations, property initializers need not be compile-time constants; however, a given compiler plugin may do additional checks to constrain the allowable initializer expressions. The X10 type-checker does not check that all properties of an annotation are initialized, although this could be enforced by a compiler plugin.

$$\begin{tikzcd}[row sep=small]
\boxed{\text{X10 source code}} \arrow[d] \\
\text{parser} \arrow[d] \\
\boxed{\text{annotated X10 AST}} \arrow[d] \\
\text{base X10 type-checker} \arrow[d] \\
\boxed{\text{annotated, type-checked AST}} \arrow[d] \\
\text{compiler plugins} \arrow[d] \\
\boxed{\text{transformed AST}} \arrow[d] \\
\text{compiler backend} \arrow[d] \\
\boxed{\text{target code}}
\end{tikzcd}$$

Figure 1: Annotation processing and compiler plugin architecture

2.1 Using annotations

Annotations can be applied to most syntactic constructs in the language including class declarations, constructors, methods, field declarations, local variable declarations and formal parameters, statements, expressions, and types.

Annotations are declared as X10 interfaces and consequently live in the same namespace as classes and interfaces. To use annotations, the programmer can import them into the compilation unit or can use their fully qualified names.

X10 provides several annotations in the `x10.lang` package, which is imported into all compilation units. X10 also provides syntactic sugar for some annotations.

2.2 Annotation syntax

The annotation syntax consists of an "`@`" followed by an interface type.

```
Annotation  ::= @ InterfaceType
Annotations ::= Annotations Annotation
              | /* empty */
```

Recall that the X10 `InterfaceType` syntax is:

```
InterfaceType    ::= TypeName DepParametersopt
                         PlaceTypeSpecifieropt
DepParameters    ::= ( DepParameterExpr )
DepParameterExpr ::= ArgumentList WhereClauseopt
WhereClause      ::= : Constraint
Constraint       ::= Expression
ArgumentList     ::= Expression
                   | ArgumentList , Expression
```

In the current version of X10, the parameters and constraint must be immutable. We relax the base X10 v1.0 type system to allow arbitrary dependent expressions to appear in interface types. A compiler plugin may restrict the syntax to immutable expressions.
The following are syntactically legal annotations:

```
@nonnull

@range(1,2)

@where(x < y)
```

2.3 Type-checking annotations

Annotations are type-checked as normal X10 interface types. It is a compile-time error if the annotation does not resolve to a valid interface type in the base type system. A compiler plugin may implement additional checks to ensure it can process the annotation. For instance, it may require that certain properties be defined or that property initializers are compile-time constants.

Since annotations are types, property initializers are not evaluated at run time. However, plugins may generate code using the property initializers, for example to implement casts. Since the initializers have no semantic meaning in the base type system, they are not evaluated at compile time by X10 type checker; however, they may be interpreted by compiler plugins.

Annotations are type-checked in the context containing the annotation, augmented as follows:

- The special variable `@self` refers to the annotation itself. Instance methods and properties declared in the annotation may be accessed through `@self`.
- For type annotations on a given type, the special variable `self` has the unannotated type.
- The context of a class annotation includes the special variable `this` of the class type.
- The context of a method annotation, a constructor annotation, a formal parameter annotation, a formal parameter type annotation, a return type annotation, a `throws` type annotation, or a `this` type annotation appended after the formal parameters includes all formal parameters of the method or constructor and also the special variable `this`, if the method is not `static`.
- For a non-`static` field, the context of a field annotation or a field type annotation includes the special variable `this`.

Unqualified identifiers in property initializers are resolved with rules similar to those for X10 constraints, except names are looked up in the scope of `@self` as well as in the scopes of `self` and `this`.

2.4 Annotation placement

Annotations may be placed on declarations, types, statements, and expressions. Multiple occurrences of the same annotation (i.e., multiple annotations with the same interface type) on the same entity are permitted.

2.4.1 Declaration annotations

Type declarations, constructors, methods, field declarations, local variable declarations, and formal parameters can be annotated by adding an annotation among the access modifiers.

```
Flag ::= public | private | ...
       | Annotation
```

The following examples illustrate the syntax:

```
@Override
public void add(Object o) { ... }

@PreCondition(0 <= i && i < this.size)
public Object get(int i) { ... }

@lent A x;
```

2.4.2 Type annotations

Types are annotated by appending annotations to the type.

```
Type ::= ...
       | Type Annotations
```

The following examples illustrate the syntax:

- Class and interface types:

  ```
  Object@nonnull
  ```

- Primitive types:

  ```
  int@range(1,4)
  ```

- Array types:

  ```
  double[][]@size(n * n)

  double[.]@rank(2)@size(n * n)
  ```

- Closure types[1]:

  ```
  boolean(T,T) @symmetric
  ```

In JSR 308 [9], type annotations are prepended rather than appended. However, this introduces a parsing ambiguity on declarations. By appending the annotations the ambiguity does not occur:

```
@anno T x; // annotation on the variable x
T@anno x;  // annotation on the type T of x
```

[1]Closures are a work-in-progress. There is a proposal for adding closures to X10 at [13].

The authors of JSR 308 are aware of the ambiguity and site backward compatibility with existing annotations and potential confusion about where annotations should be placed.

Types annotations may be placed in almost any context in which a type occurs:

- Casts:

```
y = (int@range(1,4)) x
```

- `instanceof` expressions:

```
str instanceof String@nonnull
```

- `new` expressions:

```
List@nonempty list;
new List@nonempty(list);
```

- Inheritance:

```
class ReadonlyList implements List@readonly { ... }
```

- Variable declarations:

```
Object@nonnull x;
```

- Formal parameters:

```
void set(Object@nonnull x) { ... }

Collection(List<T>@nonnull x) { ... }

try { ... }
catch (Exception@critical e) { ... }
```

- Return types:

```
int@where(self >= 0) size();
```

- `throws` clauses:

```
void work() throws WorkError@critical { ... }
```

The interface type occurring in an annotation *cannot* itself be annotated because it is syntactically ambiguous; that is `@a1@a2` is parsed as two annotations not as one annotation, `a1` annotated with `a2`.

Since the type of the special variable `this` is implicit in member declarations, it requires special syntax. Method receivers may be annotated by placing the annotation after the formal parameters. For example:

```
Object get(Object key) @ReadOnly { ... }
```

The constructor return type may be annotated by placing the annotation after the class name. For example:

```
C@initialized(int x) { ... }
```

Field targets may be annotated by placing the annotation after the field name. For example:

```
String file @initialized(this);
```

2.4.3 Statement annotations

Statements can be annotated by prepending an annotation:

```
Statement ::= Annotation Statement
```

For example:

```
@local { ... }

@atomic { ... }

// declassify program counter label to label L
@declassify(L) { ... }

// annotate with profile information
if (c)
  @probability(90) S1;
else
  @probability(10) S2;
```

Annotated empty statements (;) can appear to be statements in their own right. for example:

```
@assert(c); // statically check that c holds here
```

2.4.4 Expression annotations

Expressions can be annotated using a cast-like syntax:

```
Expression ::= ( Annotation ) Expression

(@remote) m()      // rpc

(@usually(0)) a[i] // value profiling
```

3 Annotation declarations

Annotations are declared as normal X10 interfaces. Annotations may therefore have properties, static fields, and instance methods. For example, a non-null annotation on types might be declared:[2]

```
// annotate subtypes of T
interface TypeAnnotation(Type T) { }

interface Nullable
  extends TypeAnnotation(x10.lang.Object) { }
```

A range annotation might be declared:

```
interface Range(int lo, int hi)
  extends TypeAnnotation(int) { }
```

[2]We assume X10 supports generics using Type properties. A design for generics is currently in progress.

Like interfaces, an annotation may extend one or more other annotations. The derived annotation inherits all members of its base annotations. Since annotations are interfaces, they do not have constructors. An annotation declaration may instantiate the properties of its base annotations in the `extends` clause. For example:

```
interface where(Type T, boolean constraint)
  extends TypeAnnotation(T) { }

interface rank(int rank)
  extends where(region, self.rank == rank) { }
```

Since they are normal interface declarations, annotation declarations may also contain instance methods. New operations to be used in property initializers can be declared as annotation interface methods and can be invoked through `@self`.

For example, the following annotation provides a ∀ operation through a method named `forall`. In this case, the implementation of the operation is provided by the plugin, not by user code. The compiler plugin interprets calls to `forall` as the ∀ quantifier, and the `forall` method declaration is used for type-checking these ∀ expressions in user code.

```
interface where(Type T)
  extends TypeAnnotation(T) {

  boolean forall(Collection(T) c,
                 boolean(T) test);
}

boolean(int) pos = boolean(int x) { x >= 0; }
Set(int)@where(self != null &&
               @self.forall(self, pos));
```

4 Built-in annotations

X10 provides several built-in annotations in the package `x10.lang`.

4.1 Infrastructure annotations

The following annotations support the annotation system itself.

- `PluginMandatory` may be applied to any interface declaration. The annotation `@PluginMandatory(true)` specifies that it should be a compile-time error if no plugin is found for the interface if the interface is used in as an annotation.

  ```
  interface PluginMandatory(boolean required)
    extends ClassAnnotation { }
  ```

  Most built-in annotations are mandatory. Annotation declarations not marked with a `PluginMandatory` annotation are optional; no plugin for these annotations need be invoked.

- `PluginClass` may be applied to any interface declaration. The annotation `@PluginClass(Name)` specifies the compiler plugin named `Name` should be used to process the annotation.

```
@PluginMandatory(true)
interface PluginClass(String pluginName)
  extends ClassAnnotation { }
```

- `TypeAnnotation` may be extended by an annotation on types. The plugin for `TypeAnnotation` checks that any annotation that is a subtype of `TypeAnnotation(T)` occurs only on types that are a subtype of `T`. Annotations that subtype `TypeAnnotation` with the parameter unbound may be applied to any type. Programs where a `TypeAnnotation` is applied to a declaration, statement, or expression are rejected.

```
@PluginMandatory(true)
interface TypeAnnotation(Type(+) T) { }
```

- `ExpressionAnnotation` may be extended by annotations on expressions. The plugin checks that subtypes of `ExpressionAnnotation(T)` occur only on expressions of static type `T` and that subtypes of `ExpressionAnnotation` with the type unbound occur only on expressions of arbitrary type.

```
@PluginMandatory(true)
interface ExpressionAnnotation(Type T) { }
```

- `StatementAnnotation` may be extended by annotations on statements. The plugin checks that subtypes of `StatementAnnotation` occur only on statements.

```
@PluginMandatory(true)
interface StatementAnnotation { }
```

- `ClassAnnotation` may be extended by annotations on class or interface declarations. The plugin checks that subtypes of `ClassAnnotation` occur only on class or interface declarations.

```
@PluginMandatory(true)
interface ClassAnnotation { }
```

- `MethodAnnotation` may be extended by annotations on method declarations. The plugin checks that subtypes of `MethodAnnotation` occur only on method declarations.

```
@PluginMandatory(true)
interface MethodAnnotation { }
```

- `VariableAnnotation` may be extended by annotations on variable declarations. The plugin checks that subtypes of `VariableAnnotation` occur only on local variable declarations, formal parameters, and field declarations.

```
@PluginMandatory(true)
interface VariableAnnotation { }
```

- `FieldAnnotation` may be extended by annotations on field declarations. The plugin checks that subtypes of `FieldAnnotation` occur only on field declarations.

```
@PluginMandatory(true)
interface FieldAnnotation
  extends VariableAnnotation { }
```

Sugar	Expansion
`T(:c)`	`T@where(T, c)`

`C(:c1)(T1 x1, ..., Tn xn : c2) {... }`	`@whereStatic(c2)`
	
	 `C@where(C, c1)(T1 x1, ..., Tn xn) {... }`

`this(:c) T f;` (in class C)	`T f @where(C, c);`

`this(:c1) T m(T1 x1, ..., Tn xn : c2) {... }`
 (in class C)
	`@whereStatic(c2)`
	
	 `T m(T1 x1, ..., Tn xn)@where(C, c1) {... }`

`class C(T1 x1, ..., Tn xn : c) {... }`	`@whereStatic(c)`
	
	 `class C(T1 x1, ..., Tn xn) {... }`

`T!p`	`T@where(:self.location == p)`

Figure 2: Dependent type syntactic sugar

4.2 Dependent constraint annotations

The following annotations are used to implement X10 v1.0 dependent type constraints.

```
@PluginMandatory(true)
interface where(Type T, boolean constraint)
  extends TypeAnnotation(T) { }

@PluginMandatory(true)
interface whereStatic(boolean constraint) { }
```

To process `where` annotations, the X10 type-checker checks that the constraints in the `where` annotations are type-correct in the base X10 type system. The plugin for `where` annotations then checks that the constraints are in the subset of the X10 expression syntax that it can process, constructs constraints, and finally checks the constraints.

Multiple `where` annotations on the same type are equivalent to the conjunction of the `where` constraints; that is,

```
T@where(c1)@where(c2)...@where(cn)
```

is equivalent to:

```
T@where(c1 && c2 && ... && cn)
```

The syntactic sugar in Figure 2 is provided for readability and for backward compatibility with X10 v1.0.

4.3 Other annotations

The following additional annotations are also built in:

```
// The statement is local
// sugar: local S = @Local S
@PluginMandatory(true)
interface Local extends StatementAnnotation { }

// The statement is atomic
// sugar: atomic S = @Atomic S
@PluginMandatory(true)
interface Atomic extends StatementAnnotation { }

// The methods is atomic
// sugar: atomic T m(...) =
//        @AtomicMethod T m(...)
@PluginMandatory(true)
interface AtomicMethod
  extends MethodAnnotation { }

// The method is non-blocking
// sugar: nonblocking T m(...) =
//        @Nonblocking T m(...)
@PluginMandatory(true)
interface Nonblocking
  extends MethodAnnotation { }

// The type is nullable
// sugar: nullable T = T @Nullable(T)
@PluginMandatory(false)
interface Nullable(Type T)
  extends TypeAnnotation(T) { }

// Unsafe arrays
// sugar: new T unsafe[e] = new T[e]@Unsafe(T[])
@PluginMandatory(true)
interface Unsafe(Type T)
  extends ExpressionAnnotation(T) { }
```

5 Examples

5.1 Memory models

Specifying the memory model as part of the language design can tie the semantics to particular architectures or implementations. Using annotations instead leaves the memory model open-ended, allowing more flexibility for future changes and extensions, and enabling experimentation with different models.

For example, the Java memory model, defined in the original Java Language Specification [7], was found to over-constrain implementations of compiler and virtual machines, preventing many optimizations [12]. These problems were addressed in Java 5 [8, 10].

An alternative approach is to implement the memory model using annotations, compiler plugins, and libraries. One example of this approach is OpenMP [5], which uses compiler pragmas, to declare variables as shared or private, to label code that should be run in parallel, or to specify ordering constraints.

As an example, a statement might be marked as a memory barrier using the following annotation:

```
interface Barrier extends StatementAnnotation { }
```

This annotation can be used as follows:

```
Globals.result = e;

// ensure result is seen by other threads
// before they see done set to true.
@Barrier;

Globals.done = true;
```

5.2 Nullable types

The `Nullable` annotation is used to mark object types that are allowed to be `null`. For example, the statements below are handled by the plugin as follows:

```
C@Nullable x;
C y;
x = null;              // ok
y = null;              // compile-time error
x = y;                 // ok
y = x;                 // compile-time error
y = (C@Nullable) null; // run-time error
```

The annotation is specified with the following interface:

```
interface Nullable
  extends TypeAnnotation(x10.lang.Object) { }
```

The plugin type-checks the program, prohibiting assignments from type `T@Nullable` to `T`.

5.3 Ranges and bounds checks

The `Range` annotation statically bounds the values stored in an `int` and can be used for static array bounds checking.

```
interface Range(int hi, int lo)
  extends TypeAnnotation(int) { }
```

For example:

```
int@Range(1,10) x = 11;         // rejected

final T[] a = new T[n];
int@Range(0,a.length-1) x = 11; // rejected
int@Range(0,a.length-1) x =
  (int@Range(0,a.length-1)) 11; // run-time check
```

```
int@Range(0,a.length-1) i;

// ok
for (i = 0; i < a.length; i++) {
    a[i];
}

// rejected: i goes out of bounds
for (i = 0; i <= a.length; i++) {
    a[i];
}
```

When the X10 compiler generates native code, the plugin for `Range` can be used to eliminate run-time bounds checking.

6 Compiler plugins

This section sketches an implementation of compiler plugins for the current X10 compiler, based on the Polyglot compiler framework [11].

Plugins are implemented as Polyglot passes applied to the AST after normal base X10 type checking. Plugins to run are specified on the command-line. The order of execution is determined by the Polyglot pass scheduler.

Unlike plugins for Java annotations, X10's plugins can perform code transformations, including generating code for checking annotations at run time. Thus, a plugin can implement annotations with run-time semantics. The annotations implemented by these plugins should be declared mandatory using the `PluginMandatory` annotation. Java annotations are restricted to plugins with no run-time semantics.

Because plugins may transform the AST and may modify the type system, they may generate incorrect code. In general, plugins are responsible for ensuring program transformations implement the desired program semantics and preserve the well-formedness of the AST and the type system. Plugins should produce ASTs that type-check in the base type system. Optionally, the base X10 semantic checking passes can be rerun on the generated AST to assure that later translation passes do not fail and that the generated target code will not have run-time or compile-time errors.

Plugins and annotations do not necessarily correspond. An annotation may be processed by any number of plugins: zero, one, or more than one. A plugin should not remove annotations from the AST to allow later plugins to process the annotations. A plugin may also process no annotations at all.

The base language compiler includes two goals for each job, `PluginsLoaded` and `PluginsCompleted`. `PluginsLoaded` is responsible for installing plugin dependencies into the compiler schedule. `PluginsCompleted` acts as a barrier to ensure all plugin passes are completed before code generation.

`PluginsLoaded` is a prerequisite of `PluginsCompleted`. The pass implementing `PluginsLoaded` traverses the AST. When an annotation site is encountered, the plugin for the annotation is loaded, as described in Section 6.1. The plugin exports a `Goal` object, which is made a prerequisite of the `PluginsCompleted` goal. The plugin `Goal` may depend on other `Goal`s, including those of other annotations. `PluginsCompleted` is a prerequisite of the first translation goal that rewrites X10 code to Java.

6.1 Loading plugins

To load a plugin for an annotation, a `PluginsLoaded` pass maps the annotation name to a plugin factory. which is loaded via reflection. The plugin factory returns a goal for the given plugin via the following interface:

```
package polyglot.ext.x10.plugin;

interface PluginFactory {
  polyglot.frontend.Goal getPluginGoal(
      polyglot.frontend.ExtensionInfo ext,
      polyglot.frontend.Job job, String name);
}
```

If the annotation was declared with `@PluginClass(Name)`, then the plugin factory class is `Name`. Otherwise, when the compiler sees an annotation declaration, `p1. ... .pn.A`, it will attempt to load the plugin factory using each of the following class names, in turn, until a factory with a goal for the annotation can be found:

```
p1. ... .pn.A.PluginFactory
p1. ... .pn.PluginFactory
p1.p2.PluginFactory
p1.PluginFactory
```

For each factory, the `PluginsLoaded` pass calls `getPluginGoal` with the fully qualified annotation name. Once a non-null `Goal` is returned, the `Goal` is added to the schedule.

6.2 Plugin library

We will provide an extensible framework for writing plugins. While plugins are unconstrained—they may perform arbitrary program transformations—they are also difficult to write. Plugins derived from the framework are more limited, but are easier to implement.

The plugin framework includes classes that implement the following:

- read-only passes over the abstract syntax tree,
- simple type qualifiers such as `Nullable`,
- program transformations needed to implement casts and `instanceof`,
- variable substitution for annotation properties, to facilitate plugins that use dependent types.

The details of the framework are still to be worked out. We expect the framework to evolve as more plugins are implemented.

6.3 Plugin interface

Plugins are given access to a Polyglot AST and type system. Annotations are represented in the AST as `Nodes` with the following interface:

```
package polyglot.ext.x10.ast;

interface AnnotationNode extends Node {
  ClassType annotation();
}
```

The `Node` interface is extended with the following method:

```
List<AnnotationNode> annotations();
```

In the type system, annotations are represented by the following interface:

```
interface Annotation extends TypeObject {
  ClassType annotation();
}
```

The interface `TypeObject` is extended with the following method:

```
List<Annotation> annotations();
```

7 Related work

Pluggable and optional type systems were proposed by Bracha [2]. The idea is to completely separate the run-time semantics from the type system. Type annotations, implemented in compiler plugins, serve only to reject programs statically that might otherwise have dynamic type errors.

Java version 5 [8] supports annotations on declarations. Annotations must be compile-time constants, limiting their expressiveness. In addition, plugins may not perform code transformations. Java annotations can be compiled into bytecode annotations and can exist at run time; this functionality can be implemented with compiler plugins in our system using code transformations. Recently, JSR 308 [9] has been proposed to extend Java with annotations on types. Our design provides a richer annotation syntax, as well as annotations on expressions and statements.

JavaCOP [1] is a pluggable type system framework for Java. Annotations are defined in a meta language that allows type-checking rules to be specified declaratively. Since these rules can only further restrict legal Java programs and cannot specify code transformations, JavaCOP is not as flexible as our system for X10. However, the meta language permits these extensions to be implemented concisely without knowledge of the compiler internals.

CQual [6] extends C with user-defined type qualifiers, similar to annotations on types in X10. These qualifiers may be flow-sensitive and may be inferred. Both of these features can be implemented in compiler plugins for X10, and the proposed plugin framework provides abstract plugins for qualifiers. CQual supports only a fixed set of typing rules for all qualifiers. In contrast, the *semantic type qualifiers* of Chin, Markstrum, and Millstein [4] allow programmers to define typing rules for qualifiers in a meta language. The rules are checked for soundness.

8 Conclusions

This paper presents a design for annotation system and compiler plugin facility for X10. Annotations are statically checked at compile time by the base X10 type-checker.

Writing plugins requires rather intimate knowledge of the X10 compiler implementation. This is somewhat alleviated by the plugin framework. In the future, we plan to provide a meta language for writing annotation checkers and program transformations. This language would make plugins independent of the details of the compiler in which the plugin is embedded. The meta language could itself be implemented as a plugin in the current framework.

Acknowledgments

Michael Adams, Vincent Cave, Robert Fuhrer, Igor Peshansky, Lex Spoon, Tom Schrijvers, Pradeep Varma, and Krishna Nandivada Venkata provided useful feedback on the design and presentation.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed

in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

References

[1] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSPLA)*, October 2006.

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

[3] Philippe Charles, Christian Grothoff, Christopher Donawa, Kemal Ebcioğlu, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSPLA)*, October 2005.

[4] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the 20th ACM SIGPLAN Conference Programming Language Design and Implementation (PLDI))*, pages 85–95, 2005.

[5] L. Dagum and R. Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January 1998.

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

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1st edition, 1996.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005. ISBN 0321246780.

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

[10] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2005.

[11] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag.

[12] William Pugh. The Java memory model is fatally flawed. *Concurrency—Practice and Experience*, 12(6):445–455, May 2000.

[13] Vijay Saraswat. Proposal for closures in X10. `http://vjsaraswat.blogspot.com`, July 2006.

[14] Vijay Saraswat et al. Report on the experimental language X10. `http://x10.sourceforge.net/docs/x10-101.pdf`, December 2006. Version 1.01.