The Design and Implementation of Data-Centric Sychronization for Structured Parallel Program

Sai Zhang (szhang@cs.washington.edu)

September 26, 2011

Abstract

This document describes the design and implementation details of supporting data-centric sychronization for the X10 programming language. It includes: (1) a detailed description of the current design and implementation, and (2) tricks, tips and commons pitfalls in modifying the X10 compiler infrastructure for future extensions.

1 Current Status

As of Sep 2011, the implementation fully supports the initial design as described in Section 2. However, the current implementation *only* supports Java backend. It has been tested on 6 small examples from the X10 release package plus 1 medium-size subject called *fmm* from the anuchem application (`http://cs.anu.edu.au/ Josh.Milthorpe/anuchem.html`).

The source code and tested programs can be found at the following svn repository: `http://x10.svn.sourceforge.net/viewvc/x10/branches/atomic-sets/`.

The following sections will illustrate the implemented data-centric sychronization features (Section 2), all major implementation details (Section 3), and a few summarized tricks, tips and common pitfalls in modifying the compiler for future extensions (Section 4).

2 Data-centric Sychronization for X10

This sections gives a high-level overview of what has been implemented. Most of the content is summarized from emails between Mandana Vaziri and Sai Zhang.

2.1 Programming model

The programming model consists of declaring a collection of data as being part of a data group, and indicating the transactional boundaries for that data group (or unit of work, atomic section). A unit of work is such that the elements of the data group to which it belongs are manipulated atomically. The units of work provide mutual exclusion for accesses to elements of the data group.

Each instance of a class has a single implicit data group labeled this, and all fields of a class belong to this data group. A data group may contain only data that is located in the same X10 place.

Units of work are indicated using the construct `atomic(`var_1`, ..., `var_n`) { ... }`, which means that the object referenced by var_i is manipulated atomically, if var_i is an object reference, and that var_i is accessed atomically if var_i is of primitive type. More formally, if var_i refers to an object, `atomic(`var_i`){ ... }` declares a unit of work for the data group of the object referenced by var_i. In this case, var_i may be a field, a formal parameter, or a local variable. If var_i is a primitive type, `atomic(`var_i`){ ... }` declares a unit of work for an implicitly locally defined data group that only contains var_i. In this case, var_i can only be a local variable or a primitive formal parameter.

So far atomic sections look very much like Java's synchronized block. What differentiate them is that multiple objects may share the same data group. This is indicated using the `linked` keyword on a field, formal parameter, or local variable. This keyword means that the referenced object has the same data group as the data group of 'this'. It is not allowed to label variables of primitive type with the keyword `linked`.

2.2 High-level implementation

We add to each class, an additional field that holds the lock for the data groups of instances of that class. Each class is equipped with a getter method for the lock.

Constructors are modified to take an additional lock. When the `linked` keyword is used for a constructor call, the lock for `this` is passed to the newly created object.

The construct `atomic(`var_1`, ..., `var_n`) { ... }` grabs the lock for every var_i by calling the getter method for its lock, if var_i is a reference type. If var_i is a primitive value, a local lock is declared after the declaration of the local variable var_i and this lock is grabbed.

3 Implementation Details

This sections describes the most important implementation details. Section 3.1 gives an overview of the X10 compiler workflow and which parts have been modified to implement data-centric synchronization. Section 3.2 shows how is the new syntax added to the existing compiler framework (lexer and parser parts). Section 3.3 illustrates how to perform type checking in the presence of the data-centric features, and Section 3.4 presents how the X10 compiler performs code generation. Changes to the runtime library are described in Section 3.5, and limitations and possible improvement space is summarized in Section 3.6.

3.1 Code structure and Compiler workflow

All code changes reside in two sub-projects `x10.compiler` and `x10.runtime`. The project `x10.compiler` also contains a modified version of polyglot, which is under the package `polyglot`.

In `x10.compiler`, code for adding new syntax is in the `x10.parser` package. Code for type-checking is in the `x10.ast` and `polygloat.ast` packages. More specifically, class `x10.visit.X10AtomicityChecker` is the entry class for type-checking , and class `x10.visit.X10LockMapAtomicityTranslator` performs code generation. All changes to the runtime library are in the package `x10.util.concurrent`.

Configuration option: the only new configuration option is `DATA_CENTRIC` (with default value `true`) in class `x10.Configuration`. Setting this value to `false` will turn off the data-centric sychronization features. **Note:** when compiling the x10 runtime lib using command `ant dist`, this flag **must** be turned off; since the current data-centric implementation lacks the lack of C++ backend support.

Here is the general work flow of compiling an X10 program:

1. Parse the X10 program text into AST tokens. The parser is automatically generated based on the grammar files located in package `x10.parser`. When a rule defined in the `x10.g` grammar file is matched, the parser will invoke corresponding action method in class `x10.parser.X10SemanticRules` to create AST nodes. At this stage, almost AST nodes do not contain any type information. The parser merely uses a `polyglot.parse.ParsedName` object to represent each type node.
2. Translate a `ParsedName` object into a `TypeNode`. This step is done by the `ParsedName.toType` method. This method creates a `AmbTypeNode` node for each `ParsedName` object to represent a type node. However, the created `AmbTypeNode` still needs to be dis-ambiguated before type-checking.
3. Dis-ambiguate each `AmbTypeNode` node. This step is performed by the class `x10.visit.X10TypeChecker`. Method `X10TypeChecker.leaveCall` first calls `disambiguate(tc)` before performing type-checking. The `disambiguate` method is overriden in every ambiguous type node to resolve ambiguity and infer types. Thus, if a new AST node type is added, be aware of overriding the `disambiguate` method.
4. After disambuigiating each AST node, the following compiler workflow is essentially applying a set of passes to the AST tree. Each pass is implemented as a visitor. All visitor classes are under packages `x10.visit` and `polyglot.visit`. A visitor can manipulate each AST node, delete, or add needed information to it. The visiting order of each visitor is defined in class `x10.ExtensionInfo` (and `x10c.ExtensionInfo` and `x10cpp.ExtensionInfo` for Java and C++ specific passes). A good example to refer is the `goals(Job)` method.
5. After type-checking, many optimization and code generation tasks in X10 is implemented as a X10-to-X10 source-code-level transformation. A good example is the `x10.visit.Lowerer` class. When implementing new features, normally you only need to define a similar visitor (as the `Lowerer` class) to perform X10-to-X10 source transformation instead of modifying the backend translation from X10 to Java (C++).

6. The last step is generating native code (Java bytecode and C++ binary code) from the X10 AST. Normally, you do not need to touch this phase.

3.2 Adding new syntax

The syntax changes to the X10 language are:

- Add a new keyword `linked` as a type modifier.
- Add a new rule for type identifier: *Type* = `linked` *Type*. This permits programmers to link the atomic set of a variable to the current `this` atomic set by declaring: `var a:linked A = new linked A()`.
- Add a new rule for atomic section: `atomic`(*identifier_list*) `statement`. The *identifier_list* is for programmers to specify which atomic sets need to be protected. For example, `atomic (var1, this, formal1) { ... }` indicates that atomic sets to which `var1`, `this`, and `formal1` belongs are updated atomically in the atomic section.

Here are the detailed steps in implementing the above syntax changes:
Adding the `linked` keyword

1. Go to `X10KWLexer.gi` file, and modify two places. First, add `linked` as keyword by adding a new entry under the *%Export* declaration. Then, add a new entry for the `linked` modifier by adding a new entry to the *%Rules* declaration.
2. Go to `polyglot.types.Flags` class. Add a new static field declaration like `public static final Flags LINKED = createFlag("linked", null)`. Then, add three corresponding methods: `Flags linked()`,`Flags clearLinked()`, and `boolean isLinked`.
3. Go to class `x10.parser.X10SemanticRules.FlagModifier`. Add a new field declaration: `public static int LINKED = 19`, and change the field `NUM_FLAGS` correspondingly. Change the `FlagModifier.flags()` method by adding an extra if condition like `if(flag == LINKED) return Flags.LINKED`. Add a new entry in method `FlagModifier.name` like `if(flag = LINKED) return "linked"`. Finally, add a new rule for the modifier `linked` in the `X10SemanticRules` class:

   ```
   void rule_Modifier13() {
   setResult( new FlagModifier(pos(), FlagModifier.LINKED));
   }
   ```
4. Depending on how the new keyword should be used, you may also need to modify a few declarations in classes `FlagModifier` and `TypeSystem_c`. For the `linked` case, the new keyword can only be used to decorate a type, thus a new entry in the `typeModifiers` declaration is added.

Adding new grammar rules

1. Go to `x10.parser.x10.g` file. Add corresponding productions as well as their action methods in class `X10SemanticRules`.

2. For the `linked` keyword, first add a production rule under the *Modifier ::* = declaration, then add a production rule under the *TypeName ::=* declaration. For each added production rule, corresponding action method must be added in class `X10SemanticRules`.
3. For the new `atomic` section syntax, one additional rule needs to be added to the `AtomicStatement` declaration. Similarly, corresponding action method should be added in class `X10SemanticRules`.

3.2.1 Building the new parser

You need to first download the `lpg.generator`. The easiest way is to download it from its CVS repository (*lpg.cvs.sf.net* with *anonymous* user, and repository path: */cvsroot/lpg*). Be aware of choosing the a correct version for your environment. In my environment, I chose two projects `lpg.generator` and `lpg.generator.linux_x86_64`. The first project must be used by any version, and the second project is platform-specific.

Run the *grammar* task in the `x10.compiler/build.xml` configuration file, and remember to refresh the whole project.

3.2.2 Define new AST nodes and propagate type information

Using data-centric synchronization features, two variable can be declared as:

```
var a1:A = new A();
var a2:  linked A = new linked A();
```

The above variable `a1` and `a2` technically have different types. `a1` is a raw `A` object, and `a2` is a `linked A` object. Thus, the compiler must keep this *linked* information through the whole compiling process, propagating from the initial parsing phase to the type-checking phase to the code generation phase.

To achieve the above goal, the following changes are made (**Note**: the following changes can work, but may not be the optimal way for implementation):

1. Add a `FlagsNode flags` field to the `TypeNode_c` class. This field indicates whether a type node is linked to other's atomic set. The value of `flags` is `null` by default, and is set to `linked` if the current object is *linked* to somewhere else. **Note:** the `copy` method must be overriden or modified, to make sure this new `flags` field will also be copied.
2. Add a `FlagsNode flags` field to the `ParsedName` class to represent whether the type object is `linked` or not. The `flags` field can only be `null` (the default value) or `linked`.
3. Change the `ParsedName.toType` method. It checks, if the `flags` field is `linked`, the compiler needs to create a different AST node of type `X10AmbTypeNodeLinked_c` for it.
4. The newly added `X10AmbTypeNodeLinked_c` type represent a *linked* ambiguous AST node. Its `flags` field is set to `linked` inside method `ParsedName.toType`.
5. When dis-ambiguating a type node, the `linked` flags must be preserved and propagated correctly from the `X10AmbTypeNodeLinked_c` node. The code for

preserving the `linked` flag is in `X10AmbTypeNodeLinked_c.disambiguate`, and `X10Disamb_c.disambiguateNoPrefix` methods. The **most** important notice here is: when setting a type node as `linked`, that type node must be copied and then re-set the field value on the copied node (see the code in `X10Disamb_c.disambiguateNoPrefix` as an example). This is because *all* variables with the same type are sharing the *same* type node object; thus, a linked node must have a different object (with the same type value but an additional `linked` field).

6. Finally, the `AbstractNodeFactory_c`, `NodeFactory`, and `X10NodeFactory_c` should also be modified by adding additional factory methods to create the new `X10AmbTypeNodeLinked_c` nodes.

3.3 Performing type checking

The visitor `X10AtomicityChecker` performs type-checking for the new grammar rules. As indicated in its `leaveCall(Node, Node, NodeVisitor)` method, the type-checking is essentially invoking the `checkAtomicity` and `checkLinkProperty` methods on each AST node as the visitor traverses the whole tree.

Two methods `checkAtomicity` and `checkLinkProperty` are added to a few related places, namely, classes `Node_c`, `JL_c`, and `NodeOps`. The default behavior of these two methods are doing nothing. So, if needed, an AST node can override these two methods to check certain properties.

In general, `checkAtomicity` fetches the `linked` flags from the `TypeNode` (that is associated with some AST nodes that are translated from `X10AmbTypeNodeLinked_c`), and add *atomic context* to its type. The *atomic context* here represents the atomic set to which the declared var is linked to. To keep the *atomic context* information, I added a field `Type atomicContext` to class `X10ParsedClassType_c` to record the linked object type. This field is set inside the `checkAtomicity` method (for a few special cases, it is set inside the `typeCheck` method). After fetching the `atomicContext`, the visitor checks the linked property against the typing rules. (**Note** that, for most cases, `checkAtomicity` and `checkLinkProperty` can be merged into one method).

During type-checking, we not only need to check the type compatibility as the normal X10 type-checking does, but also need to check the consistency of the `atomicContext` field to see whether a variable is always linked to the same atomic set.

I next use a few examples to show how the type checking is performed:

1. checking assignment

```
class C {
 public def foo() {
   var a1:A = new A();
   var a2:  linked A = new linked A();
   a2 = a1;  //type check this assignment
  }
}
```

The type of `a1` is `A`, with `atomicContext = null`

The type of `a2` is `A`, with `atomicContext = C`, indicating variable `a2`'s atomic set is linked to `C.this`.atomic set.

Thus, when checking the assignment `a2 = a1`, the type checker will issue an error, saying that `a2` is linked to somewhere else, and can not be assigned to a raw object `a1` which is not linked to any other atomic set.

2. checking field access.

   Consider the following example (just for illustration purpose. we may make field as strongly private later):

```
 class C {
  var f:  linked C = new linked C();
}
 class B {
  public def foo() {
    var c1:  linked C = new linked C();
    c1.f = new C();  //type check this assignment
   }
}
```

   The type of `f` field is: `C` with `atomicContext = C`

   The type of `c1` is: `C` with `atomicContext = B`

   The type of `new linked C()` inside method `foo()` is: `C` with `atomicContext = B`

   The **tricky** part is that expression `c1.f` has type: `C` but with `atomicContext = B`, since as indicated by the typing rule the accessed field's `atomicContext` equals the receiver's `atomicContext` if both are linked.

   Thus, this assignment type checks.

3.4 Code generation

The code generation is a source-code-level X10-to-X10 code translation process. All relevant code is in class `X10LockMapAtomicityTranslator`.

The major code transformation consists of the following phases:

1. Class-level transformation:
   - Let each compiled class (interface) implement (inherit) `x10.util.concurrent.Atomic`.
   - Associate each class with a lock by inserting a unique lock id field. The lock id can be used to find the corresponding lock in a global lock map. Then, add corresponding getter method for the lock field.
   - For each constructor, create a new constructor by adding an additional lock field formal parameter, then add the new constructor to the class declaration.
     Here is one transformation example:

```
public class A {
   this() {...}
   this(v:Int) {}
}
```

⇓

```
public class A implements Atomic {
   var lockid:Int = -1;
   public def OrderedLock getOrderedLock() { return OrderedLock.getLock(lockid);}
   static var static_lockid:Int = OrderedLock.createNewLockID();
   public static def OrderedLock getStaticOrderedLock() { return static_lockid;}
   this() {...}
   this(lock:OrderedLock) {... this.lockid = lock.getIndex();}
   this(v:Int) {...}
   this(v:Int, lock:OrderedLock) {... this.lockid = lock.getIndex();}
}
```

2. Method-level transformation
   - Add additional local locks for parameters which are not associated with a lock (e.g., lib code, and primitive types)
   - Transform atomic method to acquire locks
     Here is an example:

```
public def foo(b:Array[Int]) {
    finish {
       async {atomic(b) {...update b... }}
       async {atomic(b) {...update b... }}
    }
}
```

⇓

```
public def foo(b:Array[Int]) {
    var lockid_for_b:Int = OrderedLock.createNewLockID();
    finish {
       async {atomic(b) {...update b... }}
       async {atomic(b) {...update b... }}
    }
}
```

Here is an example for atomic method (**note:** `atomic` method is the syntactic sugar of `atomic(this)`, and an atomic method will also protect the atomic sets of its formal parameters):

```
public atomic def foo(a:A) {
    ...//do something
}
```

⇓

```
public atomic def foo(a:A) {
  try{
    OrderedLock.acquireLocks(this.getOrderedLock(), a.getOrderedLock());
    ...//do something
  } finally {
    OrderedLock.releaseLocks(this.getOrderedLock(), a.getOrderedLock());
  }
}
```

3. Block-level transformation

   This phase primarily declares locks to protect local variables that are accessed inside an atomic section. Here is an example (in which the local `value` must be protected):

```
public def count() {
    var value:Int = 0;
    finish for (var i:Int = 0; i < 100; i++) async { atomic(value) value ++; }
}
```

$\Downarrow$

```
public def count() {
    var value:Int = 0;
    var local_lockid_for_value = OrderedLock.createNewLockID();
    finish for (var i:Int = 0; i < 100; i++) async { atomic(value) value ++; }
}
```

4. Atomic-section-level transformation

   This phase primarily grabs suitable locks for each atomic section. Here is an example which covers almost all locking cases:

```
public def foo(a:Array[Int], c:C) {
    var value:Int = 0;
    finish for (var i:Int = 0; i < 100; i++) async { atomic(value, a, c, this) { ... do something} }
}
```

$\Downarrow$

```
public def foo(a:Array[Int], c:C) {
    var local_lockid_for_a = OrderedLock.createNewLockID();
    var value:Int = 0;
    var local_lockid_for_value = OrderedLock.createNewLockID();
    finish for (var i:Int = 0; i < 100; i++) async {
        try {
             OrderedLock.acquireLocks( local_lockid_for_value,  local_lockid_for_a,
                c.getOrderedLock(), this.getOrderedLock());
             ...do something
        } finally {
              OrderedLock.releaseLocks(local_lockid_for_value,  local_lockid_for_a,
                c.getOrderedLock(), this.getOrderedLock());
        }
    }
}
```

5. Other transformations.

   In particular, do **remember** you must manually update the captured environment vars of `async`, `at`, `ateach`, and `athome` code block after performing transformation. Please see the `X10LockMapAtomicityTranslator.visitAsync_c` as an example.

3.5 Runtime library

Two classes are added to the `x10.runtime` project:

1. `x10.util.concurrent.Atomic`. An interface that every compiled class (interface) will implement (inherit) for data-centric sychronization.

2. `x10.util.concurrent.OrderedLock`. A class wrapping a `lock` field and a unique lock id identifier. This class contains all lock operations used in the compiler, such as `createNewLock`, `acquireLocks`, and `releaseLocks`. It also maintains a global lock map.

3.5.1 Utility methods

A few useful utility classes I added:

1. `x10.util.X10TypeUtils` contains a few utility methods for processing type information.

2. Two visitor classes: `x10.visit.AtomicLocalAndFieldAccessVisitor` and `x10.visit.X10AtomicLockLocalCollector` are used to fetch referred variables inside the atomic sections. Please see the code documentation for more details.

3. A few common error messages are organized in the `Errors` class.

3.6 Limitations and possible solutions

The section summarizes some known limitations in the current design and implementation:

1. The global lock map in class `x10.util.concurrent.OrderedLock` may lead to potential memory leak. This lock map maps an `Integer` lock id to an `OrderedLock` object. When the object associated with a lock id has been recycled, this corresponding map entry should be deleted. Furthermore, if the program is running on multiple places, there will be one copy of lock map per place. Thus, the lock map will no longer be a globally one. This will lead to problems like lock id conflicts, and how to deal with a sychronized object passed from one place to anther (which lock should be used to protect it?).

   Here are a few possible solutions. First, replace the global lock map with a `WeakHashMap`. This `WeakHashMap` maps each object to its associated lock object, so that when the (Java) object has been recycled, the corresponding entry will be automatically deleted. Second, override the `finalize` method in class `x10.lang.Object` to manually delete the corresponding entry in the lock map. The above two solutions can only be applied to Java backend, and there is still no clear solution for the C++ backend. Third, improve the lock id allocation mechanism to avoid conflict id from different places. A possible way is to combine the *place_id* with a *place-unique* integer as the lock id to ensure its global uniqueness. Another way is to use a separate service (running in a separate place) to allocate locks upon the request.

2. Arrays are not well supported in the current implementation. For example, you can not declare an array like: `var linkedArray: Array[linked C] = new Array[linked C]()`. Implementing this support requires to change a few

places. First, change class `AmbMacroTypeNode_c` to capture the `linked` modifier on the parameterized type. Second, change the `disambiguate` method in class `AmbMacroTypeNode_c` to propagate the linked information to each type node. Third, implement the `checkAtomicity` and `checkLinkProperty` methods in all array-related AST node classes like `polyglot.ast.ArrayAccess_c` and `polyglot.ast.ArrayInit_c` for type-checking.

3. The current design treats `linked` as a type modifier. This may unncessarily complicate the implementation (as seen above, multiple code places need to be changed to gurantee the *linked* information is correctly propagated). A more natural solution can be integrating the `linked` keyword seamlessly into the *constraint types* in X10. In that way, a linked var can be declared as: `var c:C{linked} = new C{linked}()`. Doing so can leverage the existing powerful constraint solver in X10 for type checking.
4. A few code issues (pure engineering improvement):
   - There are fairly code repetition in the `X10LockMapAtomicityTranslator` class. It is possible (but not easy) to reduce the code clones.
   - There are some classes and methods annotated with `@Deprecade` in the code base. Such classes and methods are not used in the current implementation, and thus can be safely removed (in certain cases, you may need to resolve all compilation errors; but that is straightford such as removing all references). The reason I still kept them is those code can be used as in experiments for comparison purpose. For example, the deprecaded class `X10MixedAtomicityTranslaotr` implements a different way of code generation. It *infers* all accessed variables inside each atomic section.
   - When visitor `X10LockMapAtomicityTranslator` adds new code (i.e., field declarations, constructors, field access) to the existing class declaration, it needs to make sure different instances of the same variable should share the same *def* object. However, this is not fully preserved in the current implementation (see Section 4). For example, as the visitor visits a `New_c` statement, it needs to add an additional lock argument value to it. However, at this point, the new constructor with the additional lock formal parameter has not been inserted to the class declaration yet. Therefore, the `visitNew_c` method need to create another `ConstructorDef` object. This created `ConstructorDef` object is different from the def object used for the new constructor (which is created when the visitor leaves the class declaration). This issue can lead to a few problems. Particularly, in the `ClosureRemover` and `X10InnerClassRemover` classes, when the constructor def is updated, the def referred by the `New_c` statement will not be updated correspondingly.I temporarily work around that problem by manually updating each constructor def.
   - A static option `compile` can be removed in class `Configuration`. This option should be set to `true` when compiling all x10 runtime library. As I found during my (incomplete) testing, this `compile` can be superseded by the `DATA_CENTRIC` option (set `DATA_CENTRIC` to `false` is sufficient for compiling x10 runtime lib). I leave it in the code base in case I missed some corner cases that need to manually set this flag.

4 Tricks, tips and pitfalls

I finally summarize a few useful tricks, tips and common pitfalls.

1. **How to test your code.** According to the standard user manual on the X10 website, you can run `ant dist` to build the whole compiler, use `x10c` to compile the code, and use `x10` to run the code. The command `ant dist` will compile all compiler code as well as the runtime lib, and cost over 4 minutes. Normally, you do not need to run this command everytime after making some changes only to the `x10.compiler` project. Instead, running `ant compiler-jar` is much more faster (completed in 5 seconds). If you are using eclipse, a more convenient way to test the compiler is to run the code directly inside eclipse as follows: (1) select an X10 class file on the explorer view, and (2) click the run → x10c launch option. The embedded `x10c` will automatically compile the *selected* X10 file. The output Java file is located in the `out` folder.
2. **How to debug your code.** Debugging X10 code is not an easy task. Here are a few tips:
   - **Use the generated Java file**. Each Java file contains line numbers in the original X10 code for the transformed Java code. Those line numbers are very useful to trace back to the original X10 file for fault localization.
   - **Pretty-print an AST Node.** There are two useful methods: `Node.prettyPrint(System.out)`, and `PrettyPrinter.printAST`. The first method prints an AST node in a text form (what you see in a code file), while the second one prints an AST node in a tree structure (you can see each Node types from the result).
3. **Add a compiler configuration option.** It is very easy: just add two static field declarations to file `x10.Configuration` class. The first one declares the option name, such as `public boolean OPTIMIZE = false`, and the second one is a `String` type that must end with a fixed suffix `_desc` as an explanation message, like `private static final String OPTIMIZE_desc = "Generate optimized code";`.
4. **Be aware of the visitor order.** Normally, all visitors override the `leaveCall` method to manipulate the AST. You can treat this method to visit a given AST in a bottom-up manner. Roughly speaking, for the code snippet below, it will visit code places in the order of *A, B, C, D, E, and F*.

```
public class C {                              //F
    public def foo() {                          //E
        finish                                    //D
            for (var i:Int = 0; i < 100; i++)   //C
                async {                             //B
                      var c:C = new C();          //A
                }
    }
}
```

5. **Examples for reference.** People who are new to x10.compiler often need to find existing code examples for reference when hacking the compiler infrastructure. Here are a few good places:

  - `x10.visit.Lowerer` contains many examples for X10 $\rightarrow$ X10 code transformation.
  - `x10.visit.X10InnerClassRemover` contains a few more advanced transformation code.
  - `x10.ast.X10MethodDecl_c.typeCheck` gives a quick idea on how type-checking is performed in X10.
  - `x10.visit.X10TypeChecker` gives a quick idea on how to write a visit to manipulate the AST.

6. **What I have changed.** In case I missed some important changes I made, please search ”data-centric” in the whole eclipse project. Normally, places that I editted are associated with comments with the above keyword.

5 Acknowlegement

This is the joint work with Mandana Vaziri and Olivier Tardieu. Igor Peshansky provided very useful guidance in hacking the x10 compiler infrastructure. David Grove, Yoav Zibin, and Nate Clinger gave insightful comments on many implementation issues.