Object Initialization in X10

Yoav Zibin David Cunningham Igor Peshansky Vijay Saraswat

IBM research in TJ Watson

yzibin | dcunnin | igorp | vsaraswa@us.ibm.com

Abstract

X10 is an object oriented programming language with a sophisticated type system (constraints, class invariants, non-erased generics, closures) and concurrency constructs (asynchronous activities, multiple places, global references). Object initialization is a cross-cutting concern that interacts with all these features in delicate ways that may cause type-, runtime-, and security- errors. This paper discusses possible designs for object initialization, and the "hardhat" design chosen and implemented in X10 version 2.1. Our implementation includes a fixed-point inter-procedural (intra-class) data-flow analysis that infers, for each method called during initialization, the set of fields that are read and those that are asynchronously and synchronously assigned.

1. Introduction

Constructing an object in a safe way is not easy: it is well known that dynamic dispatching or leaking `this` during object construction is error-prone [1, 5, 10], and various type systems and verifiers have been proposed to handle safe object initialization [3, 6, 9, 11]. As languages become more and more complex, new pitfalls are created due to the interactions between language features.

X10 is an object oriented programming language with a sophisticated type system (constraints, class invariants, non-erased generics, closures) and concurrency constructs (asynchronous activities, multiple places, global references). This paper shows that object initialization is a cross-cutting concern that interact with other features in the language. We discuss several language designs that restrict these interactions, and explain why we chose the *hardhat* design for X10.

Hardhat was termed in [5] and it describes a design that prohibits dynamic dispatching or leaking `this` (e.g., storing `this` in the heap) during construction. A hardhat design limits the user but also protects her from future bugs (see Fig. 1 below for two such bugs). X10's hardhat design is even stricter due to additional language features such as concurrency, places, and closures.

On the other end of the spectrum, Java and C# allow dynamic dispatching and leaking `this`. However, they still maintain type- and runtime- safety by relying on the fact that every type has a default value (also called zero value, which is either 0, `false`, or `null`), and all fields are zero-initialized before the constructor begins. As a consequence, a half-baked object can leak before all its fields are set. Phrased differently, when reading a final field, one can read the default value initially and later read a different value. Another source of subtle bugs is due to the synchronization barrier at the end of a constructor [8] after which all assignments to final fields are guaranteed to be written. The programmer is warned (in the documentation only!) that immutable objects (using final fields) are thread-safe only if `this` does not escape its constructor. Finally, if the type-system is augmented, for example, with non-null types, then a default value no longer exists, which leads to complicated type-systems for initialization [3, 9].

C++ is sacrifices type-safety on the alter of performance. Fields are not zero-initialized, and therefore if `this` leaks, one can read an uninitialized field. Moreover, method calls are statically bound during construction, which may result in an exception at runtime if one tries to invoke a virtual method of an abstract class (see Fig. 4 below). (Determining whether this happens is an intractable problem [4].)

A design for object initialization should have the following desirable properties:

Cannot read uninitialized fields One should not be able to read uninitialized fields. In C++ it is possible to read uninitialized fields, returning an unspecified value which can lead to unpredictable behavior. In Java, fields are zero initialized before the constructor begins to execute, so it is possible to read the default or zero value, but never an unspecified value.

Single value for final fields Final fields can be assigned exactly once, and should be read only after assigned. In Java it is possible to read a final field before it was assigned, therefore returning its default value.

Immutable objects are thread-safe Immutable classes are a common pattern where fields are final/const and instances have no mutable state, e.g., `String` in Java. Immutable objects are often shared between threads without any explicit synchronization, because programmers assume that if another thread gets a handle to an object that that thread should see all assignments done during initialization. However, weak memory models today do not necessarily have this guarantee and immutable objects could be thread-*unsafe*! Fig. 1 below will show that this can happen in Java if `this` escapes from the constructor [8].

Simple The order of initialization should be clear from the syntax, and should not surprise the user. Dynamic dispatching during construction disrupts the order of initialization by executing a subclass's method before the superclass finished its initialization. This is error-prone and often surprises the user.

Flexible The user should be able to express the common idioms found in other languages with minor variations.

Type safe The language should continue to be statically type-safe even if it has rich types that do not have a default or zero value, such as non-null types (`T{self!=null}` in X10's syntax). Type-safety implies that reading from a non-null type should never

X10'11, June 4, 2011, San Jose, California, USA.

```
class A {
  static HashSet INSTANCES = new HashSet();
  final int a;
  A() {
    a = initA(); // dynamic dispatch!
    System.out.println(toString()); //again!
    INSTANCES.add(this); // leakage!
  }
  int initA() { return 1; }
  public String toString() { return "a="+a; }
}
class B extends A {
  int b = 2;
  B() { super(); }
  int initA() { return b+42; }
  public String toString() {
    return super.toString()+",b="+b; }
  public static void main(String[] args) {
    new B(); // prints: a=42,b=0
  }
}
```

Figure 1. Two initialization pitfalls in Java: leaking `this` and dynamic dispatch during construction.

return `null`. Adding non-null types to Java [2, 3, 9] has been a challenge precisely due to Java's relaxed initialization rules.

We believe the hardhat design in X10 has these desirable properties, however they come at a cost of limiting flexibility: it is not possible to express cyclic immutable structures in X10 version 2.1. Alternative designs for initialization are described in Sec. 3, such as the `proto` design (that was part of X10 version 2.0) that allows cyclic immutable structures at the cost of a more complicated design.

The remainder of this introduction presents error-prone sequential initialization idioms in Sec. 1.1, thread-unsafe immutable objects and serialization in Sec. 1.2, and common initialization pitfalls in parallel X10 programs in Sec. 1.3.

1.1 Initialization pitfalls in sequential code

Fig. 1 demonstrates the two most common initialization pitfalls in Java: leaking `this` and dynamic dispatching. We will first explain the surprising output due to dynamic dispatching, and then the less known possible bug due to leaking `this`.

Running this code prints `a=42,b=0`, which is surprising to most Java users. One would expect `b` to be 2, and `a` to be either 1 or 44. However, due to initialization order and dynamic dispatch, the user sees the default value for `b` which is 0, and therefore the value of `a` is 42. We will trace the initialization order for `new B()`: we first allocate a new object with zero-initialized fields, and then invoke the constructor of `B`. The constructor of `B` first calls `super()`, and only afterward it will run the field initializer which sets `b` to 2. This is the cause of surprise, because *syntactically* the field initializer comes before `super()`, however it is executed after. (And writing `b=2;super();` is illegal in Java because calling super must be the first statement.) During the `super()` call we perform two dynamic dispatches: the two calls (`initA()` and `toString()`) execute the implementation in `B` (and recall that `b` is still 0). Therefore, `initA()` returns 42, and `toString()` returns `a=42,b=0`.

This bug might seem pretty harmless, however if we just change the type of `b` from `int` to `Integer`, then this code will now throw a `NullPointerException`, which is more severe.

The second pitfall is leaking `this` before the object is fully-initialized, e.g., `INSTANCES.add(this)`. Note that we leak a partially-initialized object, i.e., the fields of `B` have not yet been assigned and they contain their default values. Suppose that some other thread iterates over `INSTANCES` and prints them. Then that thread might read `b=0`. In fact, it might even read `a=0`, even though we just assigned 42 to `a` two statements ago! The reason is that this write is guaranteed to be seen by other threads only after an implicit synchronization barrier that is executed after the constructor ends. Sec. 1.2 further explains about final fields in Java and this implicit synchronization barrier.

The hardhat design in X10 (described in Sec. 2) prevents both pitfalls, because its rules prohibit leaking `this`, and they only allow calling private or final methods (that cannot be overridden). It is possible to annotate a method with `@NoThisAccess`, which allows overriding but prohibits any access to `this`. This can be useful, e.g., if you want to subclass and override a factory method to change the concrete type of the object constructed, when the original class uses the factory method in its constructor. It is possible to fix the bugs in this example by following the hardhat initialization rules in the following way: (i) Instead of leaking `this` in the constructor, we should add factory methods to create instances of `A` and `B`, and add the new fully-initialized instance to `INSTANCES` in the factory methods. (ii) We should mark `initA` as `@NoThisAccess`, and therefore it can be overridden in `B`, but it cannot access field `b`. (iii) We need to define a private or final method `toStringOnlyA`; this method cannot be overridden so it can be called during construction; the public non-final method `toString` could delegate to `toStringOnlyA`.

1.2 Final fields, Concurrency, and Serialization

We will start with an anecdote: suppose you have a friend that playfully removed all the occurrences of the `final` keyword from your legal Java program. Would your program still *run* the same? On the face of it, `final` is used only to make the *compiler* more strict, i.e., to catch more errors at compile time (to make sure a method is not overridden, a class is not extended, and a field or local is assigned exactly once). After *compilation* is done, `final` should not change the *runtime* behavior of the program. However, this is not the case due to interaction between initialization and concurrency: a synchronization barrier is implicitly added at the end of a constructor [8] ensuring that assignments to *final* fields are visible to all other threads. (Assignments to non-final fields might not be visible to other threads!)

The synchronization barrier was added to the memory model of Java 5 to ensure that the common pattern of immutable objects is thread-safe. Without this barrier another thread might see the default value of a field instead of its final value. For example, it is well-known that `String` is immutable in Java, and its implementation uses three final fields: `char[] value`, and two `int` fields named `offset` and `count`. The following code

```
"AB".substring(1)
```

will return a new string `"B"` that shares the same `value` array as `"AB"`, but with `offset` and `count` equal to 1. Without the barrier, another thread might see the default values for these three fields, i.e., `null` for `value` and 0 for `offset` and `count`. For instance, if one removes the `final` keyword from all three fields in `String`, then the following code might print `B` (the expected answer), or it might print `A` or an empty string, or might even throw a `NullPointerException`:

```
final String name = "AB".substring(1);
new Thread() {
  public void run() {
    System.out.println(name);
  }
}.start();
```

A similar bug might happen in Fig. 1 because `this` was leaked into `INSTANCES` before the barrier was executed. Consider another thread that iterates over `INSTANCES` and reads field `a`. It might read 0, because the assignment of 42 to `a` is guaranteed to be visible to other threads only after the barrier was reached.

Therefore, when creating an immutable class, Java's documentation recommends using final fields and avoid leaking `this` in the constructor. However, `javac` does not even give a warning if that recommendation is violated. X10 rules prevent any leaking of `this`, thus making it safe and easy to create immutable classes.

To summarize, final fields in Java enable thread-safe immutable objects, but the user must be careful from the pitfall of leaking `this`.

Moreover, there are two other features in Java that are incompatible with final fields: *custom serialization* and `clone`. (These two features are conceptually connected, because a clone can be made by serializing and then de-serializing.) For example, below we will explain why adding custom serialization to `String` would have forced us to remove `final` from all its fields, thus making it thread-unsafe! Similarly, these two features prevent us from adding `final` to the `header` field in `LinkedList` (even though this field is never re-assigned).

1.2.1 Custom serialization and immutability

Default serialization in Java of an object `o` will serialize the entire object graph reachable from `o`. Default serialization is not always efficient, e.g., for a `LinkedList`, we only need to serialize the elements in the list, without serializing the nodes with their `next` and `prev` pointers. (It is possible to mark a field with `transient` to exclude it from serialization. However, marking `next` and `prev` as transient would simply create nodes that are disconnected.)

Custom serialization is done by defining a pair of methods called `writeObject` and `readObject` that handle serializing and de-serializing, respectively. For example, `readObject` in `LinkedList` de-serializes the list's elements and rebuilds a new list; in that process, it assigns to field `header`. This field could have been `final` because it points to the same dummy header node during the entire lifetime of the list. However, it is re-assigned in two *methods*: `readObject` and `clone` (see Sec. 1.2.2), and final fields can only be assigned in *constructors*. It is possible to use reflection in Java to set a final field, and the new memory model (Java's spec, Section 17.5.3) even considers this:

> "In some cases, such as *deserialization*, the system will need to change the final fields of an object after construction. ... Freezes of a final field occur both at the end of the constructor in which the final field is set, and immediately after each modification of a final field via reflection or other special mechanism."

(The "other special mechanism" is default serialization that has the privilege of assigning to final fields.)

As another example, consider serializing the empty string `aVeryLongString.substring(0,0)`. The default serialization in Java will serialize the very long `char[]` with a zero `offset` and `count`. If one would have wanted to write a custom serializer for `String`, then she would have to remove the `final` keyword (making it thread-unsafe!), or use reflection to set final fields (making it inefficient). To summarize, custom serialization in Java is incompatible with final fields.

On the other hand, X10 de-serialize an object by calling a *constructor* with a `SerialData` argument (as opposed to `readObject` in Java which is a *method*). Therefore, de-serialization in X10 can assign to final fields, without using reflection and without special cases in the memory model (i.e., a freeze only happens at the end of a constructor).

Currently, the `CustomSerialization` interfaces only specify the serialize method:

```
def serialize():SerialData; // for serializing
```

There is an RFC (for X10 version 2.2 in March 2011) for adding static method- and constructor- signatures to interfaces; With that feature, the `CustomSerialization` interface would not be (partly) magical, because it will contain also the de-serialize signature:

```
def this(data:SerialData); // for de-serializing
```

The X10 compiler currently auto-generates these two methods for every class (all classes are serializable by default in X10), unless the user implemented `CustomSerialization` and wrote these two methods herself.

1.2.2 Cloning and immutability

Cloning in Java has the same incompatibility with final fields as serialization: `clone` is a method and therefore it cannot assign to final fields. However, *immutable* objects (such as strings) have no use for cloning, because you need to clone an object if you plan to *mutate* the object or the clone. Therefore, cloning is less problematic than serialization with respect to immutability.

X10 has no clone-magic as in Java. Instead, the user can (deep) clone an object using the serialization mechanism, which is invoked when a final variable is copied to another place (Sec. 1.3 explains about `at` and places in X10):

```
def clone[T](o:T) { return at (here) o; }
```

Using serialization is less efficient than directly cloning an object, and future work is planned to add cloning support to X10 that would be compatible with final fields (in a similar way as in serialization) by defining an interface `Clone` with:

```
def this(cloneFrom:CloneSource);
```

where `CloneSource` is a struct containing the target object we wish to clone.

1.3 Parallelism and Initialization in X10

X10 supports parallelism in the form of both concurrent and distributed code. Next we describe parallelism in X10 and its interaction with object initialization.

Concurrent code uses asynchronous un-named activities that are created with the `async` construct, and it is possible to wait for activities to finish with the `finish` construct. Informally, statement `async S` executes statement `S` asynchronously; we say that the newly created activity *locally terminated* when it finished executing `S`, and it *globally terminated* when it locally terminated and any activity spawned in `S` has also globally terminated. Statement `finish S` blocks until all the activities created in `S` globally terminated.

Distributed code is run over multiple *places* that do *not* share memory, therefore objects are (deeply) copied from one place to another. The expression `at(p) E` evaluates `p` to a place, then copies all captured references in `E` to place `p`, then evaluate `E` in place `p`, and finally copies the result back to the original place. Note that `at` is a synchronous construct, meaning the current activity is blocked until the `at` finishes. This construct can also be used as a statement, in which case there is no copy back (but there is still a notification that is sent back when the `at` finishes, in order to release the blocked activity in the original place). Sec. 2.11 describes how *global references* are used in distributed code to allow objects in one place to point to objects in another place.

Fig. 2 shows how to calculate the fibonacci number `fib(n)` using concurrent and distributed code. Note how `fib(n-2)` is calculated asynchronously at the next place (`next()` returns the next place in a cyclic ordering of all places), while simultaneously recursively calculating `fib(n-1)` in the current place (that will recursively spawn a new activity, and so on). Therefore, the computation will recursively continue to spawn activities at the next place until `n` is 1. When both calculations globally terminate, the `finish` unblocks, and we sum their result into the *final* field `fib`.

We note that using *final local variables* for `fib2` and `fib1` instead of fields would have made this example more elegant, however we choose the later because this paper focuses on *object* initialization. X10 has similar initialization rules for final locals and final

```
class Fib {
  val fib2:Int; // fib(n-2)
  val fib1:Int; // fib(n-1)
  val fib:Int;  // fib(n)
  def this(n:Int) {
    finish {
      async {
        val p = here.next();
        fib2 = at(p) n<=1? 0 : new Fib(n-2).fib;
      }
      fib1 = n<=0? 0 : n<=1? 1 : new Fib(n-1).fib;
    }
    fib = fib2+fib1;
  }
}
```

Figure 2. Concurrent and distributed fibonacci example. Concurrent code is expressed using `async` and `finish`: `async` starts an asynchronous activity, and `finish` waits for all spawned activities to finish. Distributed code uses `at` to shift between *places*; `here` denotes the current place. `at(p) E` evaluates expression `E` in place `p`, and finally copies the result back; any final variables captured in `E` from the outer environment (e.g., `n`) are first copied to place `p`. Possible initialization pitfall: (i) forget to use `finish`, and read from `fib2` before its write finished, (ii) write to field `fib2` in another place, i.e., `at(p) this.fib2=...`, which causes `this` to be copied to `p` so one writes to a copy of `this`.

fields, but it is outside the scope of this paper to present all forms of initialization in X10 (including local variables and static fields). Details of those can be found in X10's language specification at `x10-lang.org`.

There are two possible pitfalls in this example. The first is a concurrency pitfall, where we forget to use finish, and therefore we might read from a field before its assignment was definitely executed. Java has definite-assignment rules (using an intra-procedural data-flow analysis) to ensure that a read can only happen after a write; The hardhat design in X10 adopted those rules and extended them in the face of concurrency to support the pattern of *asynchronous initialization* where an `async` must have an enclosing `finish` (using an intra-class inter-procedural analysis, see Sec. 2.12).

The second is a distributed pitfall, where one assigns to a copy of `this` in another place (instead of assigning in the original place). Leaking `this` to another place before it is fully initialized might also cause bugs in custom serialization code (see Sec. 2.10).

The remainder of this paper is organized as follows. Sec. 2 presents the hardhat initialization rules of X10 version 2.1 using examples, by slowly adding language features and describe their interaction with object initialization. Sec. 3 describes alternative designs for object initialization (one was implemented in X10 version 2.0 and another was under consideration for 2.1), weighing the pros and cons of each. Sec. 4 summarizes previous work in the field of object initialization. Finally, Sec. 5 concludes.

2. X10 Initialization Rules

X10 is an advanced object-oriented language with a complex type-system and concurrency constructs. This section describes how object initialization interacts with X10 features. We begin with object-oriented features found in mainstream languages, such as constructors, inheritance, dynamic dispatching, exceptions, and inner classes. We then proceed to X10's type-system features, such as constraints, properties, class invariants, closures, (non-erased) generics, and structs. The parallel features of X10 allow writing concurrent code (using `finish` and `async`), and distributed code (using `at` and global references). Next we describe the inter-procedural data-flow analysis that ensures that a field is read only after it was assigned. Finally, we summarize the virtues and attributes of initialization in X10.

```
class A {
  val a:Int;
  def this() {
    LeakIt.foo(this); //err
    this.a = 1;
    val me = this; //err
    LeakIt.foo(me);
    this.m2(); // so m2 is implicitly non-escaping
  }
  // permitted to escape
  final def m1() {
    LeakIt.foo(this);
  }
  // implicitly non-escaping because of this.m2()
  final def m2() {
    LeakIt.foo(this); //err
  }
  // explicitly non-escaping
  @NonEscaping final def m3() {
    LeakIt.foo(this); //err
  }
}
class B extends A {
  val b:Int;
  def this() {
    super(); this.b = 2; super.m3();
  }
}
```

Figure 3. Escaping `this` example. **Definition of *raw*:** `this` and `super` are *raw* in non-escaping methods and in field initializers. **Definition of *non-escaping*:** A method is *non-escaping* if it is a constructor, or annotated with `@NonEscaping` or `@NoThisAccess`, or a method that is called on a raw `this` receiver. **Rule 1:** A raw `this` or `super` cannot escape or be aliased. **Rule 2:** A call on a raw `super` is allowed only for a `@NonEscaping` method. (**`final`** and `@NoThisAccess` are related to dynamic-dispatching as shown in Fig. 4.)

2.1 Constructors and inheritance

Inheritance is the first feature that interacts with initialization: when class `B` inherits from `A` then every instance of `B` has a sub-object that is like an instance of `A`. When we initialize an instance of `B`, we must first initialize its `A` sub-object. We do this in X10 by forcing the constructors of `B` to make a super call, i.e., call a constructor of `A` (either explicitly or implicitly).

Fig. 3 shows X10 code that demonstrates the interaction between inheritance and initialization, and explains by example why leaking `this` during construction can cause bugs. In all the examples, errors issued by the X10 compiler are marked with `//err`.

We say that an object is *raw* (also called partially initialized) before its constructor ends, and afterward it is *cooked* (also called fully initialized). Note that when an object is cooked, all its sub-objects must be cooked as well. X10 prohibits any aliasing or leaking of `this` during construction, therefore only `this` or `super` can be raw (any other variable is definitely cooked).

Object initialization begins by invoking a constructor, denoted by the method definition `def this()`. The first leak would cause a problem because field `a` was not assigned yet. However, even after all the fields of `A` have been assigned, leaking is still a problem because fields in a subclass (field `b`) have not yet been initialized. Note that leaking is not a problem if `this` is not raw, e.g., in `m1()`.

```
abstract class A {
  val a1:Int;
  val a2:Int;
  def this() {
    this.a1 = m1(); //err1
    this.a2 = m2();
  }
  abstract def m1():Int;
  @NoThisAccess abstract def m2():Int;
}
class B extends A {
  var b:Int = 3; // non-final field
  def this() {
    super();
  }
  def m1() {
    val x = super.a1;
    val y = this.b;
    return 1;
  }
  @NoThisAccess def m2() {
    val x = super.a1; //err2
    val y = this.b; //err3
    return 2;
  }
}
```

Figure 4. Dynamic dispatching example. **Rule 3:** A non-escaping method must be private or final, unless it has `@NoThisAccess`. **Rule 4:** A method with `@NoThisAccess` cannot access `this` or `super` (neither read nor write its fields).

We begin with two definitions: (i) when an object is *raw*, and (ii) when a method is *non-escaping*. (i) Variables `this` and `super` are raw during the object's construction, i.e., in field initializers and in non-escaping methods (methods that cannot escape or leak `this`). (ii) Obviously constructors are non-escaping, but you can also annotate methods *explicitly* as `@NonEscaping`, or they can be inferred to be *implicitly* non-escaping if they are called on a raw `this` receiver.

For example, `m2` is *implicitly* non-escaping (and therefore cannot leak `this`) because of the call to `m2` in the constructor. The user could also mark `m2` *explicitly* as non-escaping by using the annotation `@NonEscaping`. (Like in Java, `@` is used for annotations in X10.) We recommend to explicitly mark public methods as `@NonEscaping` to show intent, as done on method `m3`. Without this annotation the call `super.m3()` in `B` would be illegal, due to rule 2. (We could infer that `m3` must be non-escaping, but that would cause a dependency from a subclass to a superclass, which is not natural for people used to separate compilation.) Finally, we note that all errors in this example are due to rule 1 that prevents leaking a raw `this` or `super`.

2.2 Dynamic dispatch

Dynamic dispatching interacts with initialization by transferring control to the subclass before the superclass completed its initialization. Fig. 4 demonstrates why dynamic dispatching is error-prone during construction: calling `m1` in `A` would dynamically dispatch and call the implementation in `B` that would read the default value.

X10 prevents dynamic dispatching by requiring that non-escaping methods must be private or final (so overriding is impossible). For example, *`err1`* is caused by rule 3 because `m1` is neither private nor final nor `@NoThisAccess`.

However, sometimes dynamic dispatching is required during construction. For example, if a subclass needs to refine initialization of the superclass's fields. Such refinement cannot have any access to `this`, and therefore such methods are marked with `@NoThisAccess`. For example, *`err2`* and *`err3`* are caused by rule 4

```
class B extends A {
  def this() {
    try { super(); } catch(e:Throwable){} //err
  }
}
```

Figure 5. Exceptions example: if a constructor ends normally (without throwing an exception), then all preceding constructor calls ended normally as well. **Rule 5:** If a constructor does not call `super(...)` or `this(...)`, then an implicit `super()` is added at the beginning of the constructor; the first statement in a constructor is a constructor call (either `super(...)` or `this(...)`); a constructor call may only appear as the first statement in a constructor .

that prohibits access `this` or `super` when using `@NoThisAccess`. `@NoThisAccess` prohibits any access to `this`, however, one could still access the method parameters. (If the subclass needs to read a certain field of the superclass that was previously assigned, then that field can be passed as an argument.)

In C++, the call to `m1` is legal, but at runtime methods are statically bound, so you will get an error for calling a pure virtual function. In Java, the call to `m1` is also legal, but at runtime methods are dynamically bound, so the implementation of `m1` in `B` will read the default value of `a` and `b`.

2.3 Exceptions

Constructing an object may not always end normally, e.g., building a date object from an illegal date string should throw an exception. Exceptions combined with inheritance interact with initialization in the following way: a cooked object must have cooked sub-objects, therefore if a constructor ends normally (thus returning a cooked object) then all preceding constructor calls (either `super(...)` or `this(...)`) must end normally as well. Phrased differently, in a constructor it should not be possible to recover from an exception thrown by a constructor call. This is one of the reason why a constructor call must be the first statement in Java; failure to verify this led to a famous security attack [1].

Fig. 5 shows that it is an error to try to recover from an exception thrown by a constructor call; the reason is for the error is rule 5 that requires the first statement to be `super()`.

2.4 Inner classes

Inner classes usually read the outer instance's fields during construction, e.g., a list iterator would read the list's header node. Therefore, X10 requires that the outer instance is cooked, and prohibits creating an inner instance when the receiver is a raw `this`.

Fig. 6 shows it is an error in X10 to create an inner instance if the outer is raw (from rule 6), but it is ok to create an instance of a static nested class, because it has no outer instance.

In fact, it is possible to view this rule as a special case to the rule that prohibits leaking a raw `this` (because when you create an inner instance on a raw `this` receiver, you created an alias, and now you have two raw objects: `Inner.this` and `Outer.this`). We wish to keep the invariant that only one `this` can be raw.

In our rules, we assume that there is a single `this` reference, because we convert all inner, anonymous and local classes into static nested classes by passing the outer instance and all other captured variables explicitly as arguments to the constructor.

We now turn our attention to X10's sophisticated type-system that has features that are not found in main-stream languages: constraints, properties, class invariants, closures, (non-erased) generics, and structs.

```
class Outer {
  val a:Int;
  def this() {
    // Outer.this is raw
    Outer.this. new Inner(); //err
    new Nested(); // ok
    a = 3;
  }
  class Inner {
    def this() {
      // Inner.this is raw, but
      // Outer.this is cooked
      val x = Outer.this.a;
    }
  }
  static class Nested {}
}
```

Figure 6. Inner class example: the outer instance is always cooked. **Rule 6:** a raw `this` cannot be the receiver of `new`.

```
class A {
  val i0:Int; //err
  var i1:Int;
  var i2:Int{self!=0}; //err
  var i3:Int{self!=0} = 3;
  var i4:Int{self==42}; //err
  var s1:String;
  var s2:String{self!=null}; //err
  var b1:Boolean;
  var b2:Boolean{self==true}; //err
}
```

Figure 7. Default value example. **Definition of *has-zero*:** A type *has-zero* if it contains the zero value (which is either `null`, `false`, 0, or zero in all fields for user-defined structs) or if it is a type parameter guarded with `haszero` (see Sec. 2.8). **Rule 7:** A `var` field that lacks a field initializer and whose type has-zero, is implicitly added a zero initializer.

2.5 Constraints and default/zero values

X10 supports constrained types using the syntax `T{c}`, where `c` is a boolean expression that can use final variables in scope, literals, properties (described below), the special keyword `self` that denotes the type itself, field access, equality (`==`) and disequality (`!=`). There are plans to add arithmetic inequality (`<`, `<=`) to X10 in the future, and it is possible to plugin any constraint solver into the X10 compiler.

As a consequence of constrained types, some types do not have a default value, e.g., `Int{self!=0}`. Therefore, in X10, the fields of an object cannot be zero-initialized as done in Java. Furthermore, in Java, a non-final field does not have to be assigned in a constructor because it is assumed to have an implicit zero initializer. X10 follows the same line, and a non-final field is implicitly added a zero initializer *if its type has-zero*. Fig. 7 defines when a type *has-zero*, and gives examples of types without zero. Note that `i0` has to be assigned because it is a final field (`val`), as opposed to `i1` which is non-final (`var`).

2.6 Properties and the class invariant

Properties are final fields that can be used in constraints, e.g., `Array` has a `size` property, so an array of size 2 can be expressed as: `Array{self.size==2}`. The differences between a property and a final field are both syntactically and semantically, as seen in class `A` of Fig. 8. Syntactically, properties are defined after the class name, they must have a type and cannot have an initializer, and they must be initialized in a constructor using a property call statement written as `property(...)`. Semantically, properties are initialized before all other fields, and they can be used in constraints with the prefix `self`.

```
class A(a:Int) {
  def this(x:Int) {
    property(x);
  }
}
class B(b:Int) {b==a} extends A {
  val f1 = a+b;
  val f2:Int;
  val f3:A{this.a==self.a};
  def this(x:Int) {
    super(x);
    val i1 = super.a;
    val i2 = this.b; //err
    val i3 = this.f1; //err
    f2 = 2; //err
    property(x);
    f3 = new A(this.a);
  }
}
```

Figure 8. Properties and class invariant example: properties (`a` and `b`) are final fields that are initialized before all other fields using a property call (`property(...);` statement). If a class does not define any properties, then an implicit `property()` is added after (the implicit or explicit) `super(...)`. Field initializers are executed in their declaration order after (the implicit or explicit) the property call. **Rule 8:** If a constructor does not call `this(...)`, then it must have exactly one property call, and it must be unconditionally executed (unless the constructor throws an exception). **Rule 9:** The class invariant must be satisfied after the property call. **Rule 10:** The super fields can only be accessed after `super(...)`, and the fields of `this` can only be accessed after `property(...)`.

When using the prefix `this`, you can access both properties and other final fields. The difference between `this` and `self` is shown in field `f3` in Fig. 8: `this.a` refers to the property `a` stored in `this`, whereas `self.a` refers to `a` stored in the object to which `f3` refers. (In the constructor, we indeed see that we assign to `f3` a new instance of `A` whose `a` property is equal to `this.a`.)

Properties must be initialized before other fields because field initializers and field types can refer to properties (see initializer for `f1` and the type of `f3`). The super's fields can be accessed after the super call, and the other fields after the property call; field initializers are executed after the property call.

The *class invariant* (`{b==a}` in Fig. 8) may refer only to properties, and it must be satisfied after the property call (rule 9).

2.7 Closures

Closures are functions that can refer to final variables in the enclosing scope, e.g., they can refer to final method parameters, locals, and `this`. When a closure refer to a variable, we say that it is captured by the closure, and the variable is thus stored in the closure object. Closures interact with initialization when they capture `this` during construction.

Fig. 9 shows why it is prohibited to capture a raw `this` in a closure: that closure can later escape to another place which will serialize all captured variables (including the raw `this`, that should not be serialized, see Sec. 2.10). The work-around for using a field in a closure is to define a local that will refer only to the field (which is definitely cooked) and capture the local instead of the field as done in `closure2`.

```
class A {
  var a:Int = 3;
  def this() {
    val closure1 = ()=>this.a; //err
    at(here.next()) closure1();
    val local = this.a;
    val closure2 = ()=>local;
  }
}
```

Figure 9. Closures example. **Rule 11:** A closure cannot capture a raw `this`.

```
class B[T] {T haszero} {
  var f1:T;
  val f2 = Zero.get[T]();
}
struct WithZeroValue(x:Int,y:Int) {}
struct WithoutZeroValue(x:Int{self!=0}) {}
class Usage {
  var b1:B[Int];
  var b2:B[Int{self!=0}]; //err
  var b3:B[WithZeroValue];
  var b4:B[WithoutZeroValue]; //err
}
```

Figure 10. `haszero` type predicate example. **Rule 12:** A type must be consistent, i.e., it cannot contradict the environment; the environment includes final variables in scope, method guards, and class invariants..

2.8 Generics and Structs

Structs in X10 are header-less inlinable objects that cannot inherit code (i.e., they can *implement* interfaces, but cannot *extend* anything). Therefore an instance of a struct type has a known size and it can be inlined in a containing object. Java's primitive types (`int`, `byte`, etc) are represented as structs in X10. Structs, as opposed to classes, do not contain the value `null`.

Generics in X10 are reified, i.e, not erased as in Java. For example, instances of `Box[Byte]` and `Box[Double]` would have the same size in Java but different sizes in X10. Although generics are not a new concept, the combination of generics and the lack of default values leads to new pitfalls. For example, in Java and C#, it is possible to define an equivalent to

```
class A[T] { var a:T; }
```

However, this is illegal in X10 because we cannot be sure that `T` has-zero (see Fig. 7), e.g., if the user instantiate `A[Int{self!=0}]` then field `a` cannot be assigned a zero value without violating type-safety. Therefore X10 has a type predicate written `X haszero` that returns true if type `X` has-zero. Using `haszero` in a constraint (e.g., in a class invariant or a method guard), it is possible to guarantee a type-parameter will be instantiated by a type that has-zero.

Fig. 10 shows an example of a generic class `B[T]` that constrains the type-parameter `T` to always have a zero value. According to rule 7, field `f1` has an implicit zero field initializer. It is also possible to write the initializer explicitly (as done in field `f2`) by using the static method `Zero.get[X]()` (that is guarded by `X haszero`). Next we see two struct definitions: the first has two properties that has-zero, and the second has a property that does not have zero. According to the definition of has-zero in Fig. 7, a struct has-zero if all its fields has-zero, therefore `WithZeroValue haszero` is true, but `WithoutZeroValue haszero` is false. Finally, we see an example of usages of `B[T]`, where two usages are legal and two are illegal (see rule 12).

```
class A {
  var f1:Int;
  val f2:Int;
  val f3:Int;
  def this() {//err: f2 was not definitely assigned
    async f1 = 1;
    async f2 = 2;
    finish {
      async f3 = 3;
    }
  }
}
```

Figure 11. Concurrency in initialization example: asynchronously assign to a field. **Rule 13:** A constructor must finish assigning to all fields at least once. **Rule 14:** A final field can be assigned at most once.

```
class A {
  val f:Int;
  def this() {//err: f was not definitely assigned
    // Execute at another place
    at (here.next())
      this.f = 1; //err: this escaped
  }
}
```

Figure 12. Distributed initialization example. **Rule 15:** a raw `this` cannot be captured by an `at`.

We now turn our attention to the parallel features of X10: concurrent programming (`finish` and `async`) and distributed programming (`at` and global references). Sec. 1.3 already explained how parallel code is written in X10, and what are the common pitfalls of initialization in parallel code. Next we present the rules that prevent these pitfalls.

2.9 Concurrent programming and Initialization

Fig. 11 shows how to asynchronously assign to fields. Recall that we wish to guarantee that one can never read an uninitialized field, therefore rule 13 ensures that all fields are assigned at least once.

All three fields in `A` are asynchronously assigned, however, only `f2` is not definitely assigned. Assigning to `f3` has an enclosing `finish`, so it is definitely assigned. Field `f1` is also definitely assigned, because from rule 7 it has an implicit zero field initializer. However, field `f2` is only asynchronously assigned, and the constructor does not wait for this assignment to finish, thus violating rule 13. (The exact data-flow analysis to enforce rule 13 is described in Sec. 2.12.) Rule 14 is the same as in Java: a final field is assigned *at most* once (and combined with rule 13 we know it is assigned *exactly* once).

2.10 Distributed programming and Initialization

X10 programs can be executed on a distributed system with multiple places that have no shared-memory. Therefore, objects are copied from one place to another when captured by an `at`. Copying is done by first serializing the object into a buffer, sending the buffer to the other place, and then de-serializing the buffer at the other place. As in Java, one can write custom serialization code in X10 by implementing `CustomSerialization`, and defining the method `serialize():SerialData` and the constructor `this(data:SerialData)`.

Fig. 12 shows a common pitfall where a raw `this` escapes to another place, and the field assignment would have been done on a copy of `this`. We wish to de-serialize only cooked objects, and

```
class GlobalCounter {
  private val root = new GlobalRef(this);
  transient var count:Int;
  def inc() {
    return at(root.home) root().count++;
  }
}
```

Figure 13. A global counter example. **Revision of 1:** A raw `this` can only escape to a global ref constructor in a private field initializer; that field cannot be read via a raw `this` receiver. **Rule 16:** The type of a `transient` field must satisfy `haszero`.

therefore rule 15 prohibits `this` to be captured by an `at`. Consequently, we also report that field `f` was not definitely assigned.

2.11 Global references

A distributed data-structures is dispersed over multiple places, and it is convenient to have pointers from one place to an object in another place. These cross-places pointers are called *global references*. A global reference has two fields: `object` that points to some object, and `home` which is the place where the global reference was created. When a global reference is serialized, we serialize its `home` and the value of the *pointer* to the `object` (we do not serialize the `object`). For example, suppose that `o` is some object. Then, when a box pointing to `o` is serialized, then `o` is recursively serialized. However, when a global reference pointing to `o` is serialized, then only the pointer to `o` is serialized (not `o` itself).

A *global object* has mutable state in a single place `p` and methods that can be called from any place that mutate state in `p`. The common *global object idiom* uses a global reference to point to a single mutable object. Fig. 13 shows a global counter object that has mutable state (`count`) in a single place, but any place can increment the counter by incrementing `count` at that single place, which is `root.home`. Note how `root()` returns the referent of the global reference. (The call `root()` is guarded by `root.home==here`, which is statically verified in this code.)

Note that a raw `this` leaked into the constructor of `GlobalRef` which is a violation of rule 1. Because this idiom is common in distributed programming, we relaxed this rule and allow `this` to escape into a global ref, but only in a private field initializer that is not read during construction.

As in Java, the `transient` keyword marks a field that should not be serialized. Upon de-serialization, such a field has the zero value. Therefore, the type of a transient field must have the zero value (rule 16).

We finally note that the global object idiom is error-prone because it is easy to forget to use `root()` before accessing a mutable field. There is an RFC that suggests an annotation that will automatically convert a class to a global class using this pattern.

2.12 Read and write of fields

We now present a data-flow analysis for guaranteeing that a field is read only after it was written, and that a final field is assigned exactly once. Java performs an *intra*-procedural data-flow analysis in *constructors* to calculate when a *final* field is definitely-assigned and definitely-unassigned. In contrast, X10 performs an *inter*-procedural (fixed-point) data-flow analysis in all *non-escaping methods* to calculate when a field (*both final and non-final*) is definitely-assigned, *definitely-asynchronously-assigned*, and definitely-unassigned. The details are explained using examples (Fig. 14) by comparison with Java; the full analysis is described in X10's language specification.

X10 and Java allows *writing* to a final field only when it is definitely-*unassigned*, and it allows *reading* from a final field only

```
class A {
  val a:Int;
  def this() {
    readA(); //err1
    finish {
      async {
        a = 1; // assigned={a}
        readA();
      } // asyncAssigned={a}
      readA(); //err2
    } // assigned={a}
    readA();
  }
  // reads={a}
  private def readA() {
    val x = a;
  }
}
class B {
  var i:Int{self!=0}, j:Int{self!=0};
  def this() {
    finish {
     asyncWriteI(); // asyncAssigned={i}
    } // assigned={i}
    writeJ();// assigned={i,j}
    readIJ();
  }
  // asyncAssigned={i}
  private def asyncWriteI() {
    async i=1;
  }
  // reads={i} assigned={j}
  private def writeJ() {
    if (i==1) writeJ(); else this.j = 1;
  }
  // reads={i,j}
  private def readIJ() {
    val x = this.i+this.j;
  }
}
```

Figure 14. Read-Write order for fields. We infer for each method three sets: (i) fields it reads (i.e., these fields must be assigned before the method is called), (ii) fields it assigned, (iii) fields it asynchronously assigned. The flow maintain similar three sets before and after each statement; *`assigned`* becomes *`asyncAssigned`* after an `async`, and *`asyncAssigned`* becomes *`assigned`* after a `finish`. In the example, we omitted empty sets. **Rule 17:** A field can be read only if it is definitely-assigned. **Rule 18:** A final field can be written only it is definitely-unassigned.

when it is definitely-*assigned*. X10 also has the same read restriction on non-final fields (recall that rule 7 adds a field initializer if the field's type has-zero).

Consider first only *final* fields. They are easier to type-check because they can only be assigned in constructors. X10 extends Java rules, by calculating for each non-escaping method `m` the set of final fields it reads, and calling `m` is legal only if these fields have been definitely assigned. For example, in class `A`, method `readA` reads field `a` and therefore cannot be called before `a` is assigned (e.g., *`err1`*). Note that Java does not perform this check, and it is legal to call `readA` which will return the zero value of `a`. X10 also adds the notion of *definitely-asynchronously-assigned* which means a field was definitely-assigned within an `async` (so it cannot be read, e.g., *`err2`*), but after an enclosing `finish` it will become definitely-assigned (so it can be read). The flow maintains three sets: *`reads`*, *`assigned`*, and *`asyncAssigned`*. If a constructor reads an uninitialized field, then it is an error; however, if a method

reads an uninitialized field, then we add it to its *reads* set. Phrased differently, the *reads* set of a constructor must be empty.

Now consider non-final fields. They can be assigned and read in methods, thus requiring a fixed-point algorithm. For example, consider method writeJ. Initially, *reads* is empty, while *assigned* and *asyncAssigned* are the entire set of fields. In the first iteration, we add i to *reads*, and when we join the two branches of the if, *assigned* is decreased to only j. The fixed-point calculation, in every iteration, increases *reads* and decreases *assigned* and *asyncAssigned*, until a fixed-point is reached.

2.13 Virtues and attributes of initialization in X10

We assume there is a single this variable, because all nested classes were converted to static, as described in Sec. 2.4. Therefore, initialization in X10 has the following attributes: (i) this (and its alias super) is the only accessible raw object in scope (rule 1), (ii) only cooked objects cross places (rule 15), (iii) only @NoThisAccess methods can be dynamically dispatched during construction (rule 3), (iv) all final field assignments finish by the time the constructor ends (rule 13), (v) it is not possible to read an uninitialized field (rule 17), and (vi) reading a final field always return the same value (rule 18 combined with attribute (v)).

3. Alternative Initialization Designs

3.1 Default values design

Java first initializes fields with either 0, false, or null (depending on the field type) and then running the constructor to initialize the fields according to the programmer's wishes. If every X10 type had a default value that was statically known, then Java's object initialization scheme could be used in X10. This would have the advantage of familiarity for Java programmers that are learning X10. The disadvantages are that that it is nonintuitive that final fields can be observed to change value, and that it is prone to undetectable errors where the field is read before initialization.

Unfortunately it is hard to reconcile the notion of a default value with X10's type system, because a programmer can define a type which does not contain the default value. In the X10 type system, one can define a type with no values at all, by using a constraint that yields contradiction.

This could be addressed by extending the X10 type to require that the programmer define a new constant value for any type that has been constrained enough that the original default value is no longer a member of the type. This means every field can be initialized to the value defined in its type. The disadvantage of this is that the type system becomes more complex and more type annotations are required. We decided that this, in combination with the disadvantages given above, was too problematic to justify the advantages of Java-style object initialization.

3.2 Proto Design

If we want to allow some of the programs that the Hardhat design rejects, such as immutable cycles in the object graph, but we do not want to burden the type system with default value annotations, then one solution is to allow this to escape in certain cases while still preventing reads from uninitialized fields. This can be achieved by annotating reference types with a keyword proto to indicate that the referenced object is partially constructed. Reads of fields where the target object is proto are not allowed because a partially constructed object may not yet have initialized its fields. The advantage of this approach is that it allows a set of partially constructed objects to establish themselves as a mutually referential cycle of objects in the heap, which would not otherwise be possible. The disadvantage is that it requires an additional type annotations, although this annotation is only required in code that creates immutable cyclic heap structures. Also note that there are no additional space or runtime overheads since these extra type system mechanisms are for static checking only.

```
class C {
  val next : C {self!=null};
  var fld : C;
  def this(n:proto C{self!=null}) {
    //Console.OUT.println(n.next); //err1
    //n.f(); //err2
    this.next = n;
  }
  def this() {
    //Console.OUT.println(this.next); //err3
    //this.f(); //err4
    val c = new C(this);
    //Console.OUT.println(c.next.next); //err5
    this.next = c;
  }
  def f() {
    Console.OUT.println(this.next);
  }
  def this(c:C, Int) {
    //c.m(this); //err6
    Console.OUT.println(c.fld.next);
    this.next = c;
  }
  void m (n : proto C) proto {
    this.fld = n;
  }
  static def test() {
    val c:C{self!=null} = new C();
    assert c.next.next==c;
  }
}
```

Figure 15. An immutable cycle of heap references, using proto.

An example of an immutable cycle of two nodes is given in fig.15. A more practical but less concise example would be an immutable doubly-linked list. Let us assume that we would like to optimize away any null pointer checks, so we constrain all references to exclude the null value. The commented out lines indicate code that would be rejected by the type system.

In all constructors, this is a pointer to a partially constructed object. If the type of this were to be explicit, it would be proto C {self!=null}. The proto element of the type forbids any field reads. It also prevents the reference being leaked (e.g. into f()), except into variables of proto type where it follows that there is protection from uninitialized field reads.

The first constructor's n parameter takes a proto pointer to the original C instance. It is limited in what it can do with n, e.g. it cannot read n.next, but it can initialize its own next field with the passed-in value. When the second constructor returns, both objects are fully constructed with all fields initialized. Thus, the type of the variable c does not have a proto annotation and the field read c.next is allowed.

If a type has the proto keyword, then its fields (both var and val) may have partially constructed objects assigned to them, but fields may not be read. Conversely, the absence of proto means that the fields may be read but var fields may not have partially constructed objects assigned to them. This means that proto C and C are not related by sub-typing. In other words, proto C means definitely partially constructed and C means definitely fully constructed. Consequently it makes no sense to allow casting between the two types, and one may not extend or implement a proto type. The only sources of proto typed objects are via the this keyword in a constructor and via method parameters of proto type. The only way a type can lose its proto is by becoming fully constructed.

Consider *err5* in fig. 15. If we had inferred the type of `c` to be non-proto, then we could have read the uninitialized field `this.next`. To solve this problem, we must ensure that the whole cycle becomes fully constructed together. This can be arranged by changing the type of `new C(...)` to be `proto C` if one of its arguments is of `proto` type. This does not affect the assignment `this.next = c` because `this` is `proto`.

We do not allow fields to have `proto` type. This is because the referenced object will eventually be fully constructed and then there would be a variable of `proto` type pointing to a fully constructed instance. This admits the possibility of someone assigning a partially constructed object to a field of the fully constructed object, just as was done in the first constructor in fig. 15. Then, one could accidentally read an uninitialized field from the partially constructed object by going through the fully constructed object. Disallowing `proto` in fields avoids this problem. However local variables are safe because of lexical scoping, they will go out of scope before the constructor returns and the object becomes fully constructed.

There would be an issue calling other instance methods on `this` from a constructor, because the type of `this` in those methods would need to be `proto` since the target is still partially constructed. We support this by allowing the `proto` keyword to also be used on a method as an effect annotation, i.e. it must be preserved by inheritance. Such methods are called `proto` methods and can be called on partially constructed targets. The type of `this` then subjects the body of the method to the same restrictions as we have already seen in constructor bodies.

However in some cases we would like to avoid code duplication by allowing some methods to be callable on both `proto` and non-`proto` targets. This violates our principle that the two kinds of objects enjoy different privileges and are completely distinct. The error *err6* in fig. 15 shows how we could potentially read an uninitialized field if we allowed this relaxation.

To address this, we only allow the method to be called on non-`proto` targets if there are no `proto` parameters to the method. No such parameters means the only partially constructed object in scope is `this`. In the case where the method is called on a non-`proto` target there is therefore no partially constructed object in scope, and no harm can be done.

While we believe this type system is correct and usable for writing real programs in the X10 language, we had to decide whether the additional type system complexity and annotations were a reasonable price to pay for the additional expressiveness (i.e. the ability to construct immutable heap cycles). We ultimately decided that immutable heap cycles are too rare in practice to justify including these extra mechanisms in the language.

4. Related Work

A static analysis [10], has been used to find some default value reads in Java programs, and supports our belief that default value reads can be found in real programs and should be considered errors. Our approach is stronger (detecting all errors at the expense of some correct programs) and considers additional language constructs that are not present in Java.

There has been a study on a large body [5] of Java code, showing that initialization order issues pervade projects from the real world. A bytecode verification system for Java initialization has also been explored [6].

An early work to support non-null types in Java [2] has the notion of a type constructor *raw* that can be applied to object types and means that the fields of the object (in X10 terminology) may violate the constraints in their types. This simply disables the type-system while an object is partially constructed while ensuring the rest of the program is typed normally. Our approach prevents errors during constructors as it does not disable the type-system, and it also permits optimisation of the representation of fields whose types are very constrained, since they will never have to hold a value other than the values allowed by their type constraint.

A later work [3, 9] allows to specify the time (in the type) when the object will be fully constructed. Field reference types of a partially constructed objects must be fully constructed by the same time, which allows graphs of objects to be constructed like our `proto` design. However the system is more complicated, allowing the object to become fully constructed at a given future time, instead of at the specific time when its constructor terminates.

Masked types [9] present types that describe the exact fields that have not yet been initialized. Our type system is simpler but less expressive.

Ownership types can be used to create immutable cycles [11]. This is comparable to our `proto` design because it also allows `this` to be linked from an incomplete object. However the ownership structure is used to implement a broader policy, allowing code in the owner to use a reference to its partially constructed children, whereas we only allow code to use a reference to objects that are being partially constructed in some nesting stack frame. However our approach does not use ownership types.

There is also a time-aware type system [7] that allows the detection of data-races, and understands the concept of shared variables that become immutable only after a certain time (and can then be accessed without synchronisation). The same mechanisms can also be used to express when an object becomes cooked.

5. Conclusion

The hardhat design in X10 is strict but it protects the user from error-prone initialization idioms, especially when combined with a rich type system and parallel code. This paper showed the interaction between initialization and other language features, possible pitfalls in Java, and how they can be prevented in X10. It also presented the rules of this design, the virtues of these rules, and possible design alternatives. The rules where incorporated in the open-source X10 compiler, and are being used in production code.

References

[1] D. Dean, E. Felten, and D.S.Wallach. Java security: From hotjava to netscape and beyond. In *IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[2] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA'03*, pages 302–312.

[3] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA'07*, pages 337–350, 2007.

[4] J. Gil and A. Itai. The complexity of type analysis of object oriented programs. In *ECOOP'98*, pages 601–634, 1998.

[5] J. Y. Gil and T. Shragai. Are we ready for a safer construction environment? In *ECOOP'09*, pages 495–519, 2009.

[6] L. Hubert, T. Jensen, V. Monfort, and D. Pichardie. Enforcing secure object initialization in java. In *ESORICS'10*, pages 101–115, 2010.

[7] N. D. Matsakis and T. R. Gross. A time-aware type system for data-race protection and guaranteed initialization. In *OOPSLA'10*, pages 634–651, 2010.

[8] W. Pugh. JSR 133: JAVA memory model and thread specification revision. `http://jcp.org/en/jsr/detail?id=133`, Sept. 30, 2004.

[9] X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL'09*, pages 53–65, 2009.

[10] S. Seo, Y. Kim, H.-G. Kang, and T. Han. A static bug detector for uninitialized field references in java programs. *IEICE - Trans. Inf. Syst.*, E90-D:1663–1671, October 2007.

[11] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic java. In *OOPSLA'10*, pages 598–617, 2010.