Object Initialization in X10

Yoav Zibin[1], David Cunningham[2], Igor Peshansky[1], and Vijay Saraswat[2]

[1] Google (work done at IBM)
{yzibin,igorp}@google.com
[2] IBM research in TJ Watson
{dcunnin,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). Object initialization is a cross-cutting concern that interacts with all of 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 2.2. Our implementation includes a fixed-point interprocedural (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. Our codebase of more than 200K lines of code only had 104 annotations. Finally, we formalize the essence of initialization checking with an effect system intended to complement a standard FJ style formalization of the type system for X10. This system is substantially simpler than the masked types of [10], and it is more practical (for X10) than the free-committed types of [12]. This is the first formalization of a type and (flow-sensitive) effect system for safe initialization in the presence of concurrency constructs.

1 Introduction

Constructing an object in a safe way is not easy: it is well known that dynamic dispatch or leaking `this` during object construction is error-prone [2,11,6], and various type systems and verifiers have been proposed to handle safe object initialization [7,14,4,10]. As languages become more and more complex, new pitfalls are created due to the interactions among 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). This paper shows that object initialization is a cross-cutting concern that interacts 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 [6] is a design that prohibits dynamic dispatch or leaking `this` (e.g., storing `this` in the heap) during construction. Such a design limits the user but also protects her from future bugs (see Fig. 1 below for two such bugs). X10's hardhat design is more complex due to additional language features such as concurrency, places, and closures.

On the other end of the spectrum, Java and C# allow dynamic dispatch 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

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 207–231, 2012.

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 related to the synchronization barrier at the end of a constructor [9] 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 [4,10].

C++ sacrifices type-safety on the altar of performance: fields are not zero-initialized. (X10 has both type-safety and the performance for not zero-initializing fields.) Therefore if `this` leaks in C++, one can read an uninitialized field resulting in an arbitrary value. 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. 3b below). (Determining whether this happens is intractable [5].) We believe a design for object initialization should have these 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 among threads without any explicit synchronization, because programmers assume that if another thread gets a handle to an object, then 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*! Sec. 1.3 below will show that this can happen in Java if `this` escapes from the constructor [9].

Simple. The order of initialization should be clear from the syntax, and should not surprise the user. Dynamic dispatch during construction disrupts the order of initialization by executing a subclass's method before the superclass finished its initialization. This kind of initialization order 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 return `null`. Adding non-null types to Java [3,4,10] has been a challenge precisely due to Java's relaxed initialization rules.

We took the ideas of prohibiting dynamic dispatch or leaking `this` during construction from [6], and materialized them into a set of rules that cover all aspects of X10 (type-system, closures, generics, properties, and concurrent and distributed constructs). This hardhat design in X10 (version 2.2) has the above desirable properties, however

they come at a cost of limiting flexibility: it is not possible to express cyclic immutable structures in X10. We chose simplicity over flexibility in our design choices, e.g., X10 prohibits creating an alias of `this` during object construction (whereas a more flexible design could track aliases via alias-analysis, at the cost of sacrificing simplicity). To our knowledge, X10 is the first object-oriented (OO) language to adopt the strict hardhat initialization design.

Because one cannot read uninitialized fields in X10, there is no need zero-initialize the object's fields (as done in Java before the constructor executes). A recent study [13] measured the direct cost of *zero initialization*, which "is surprisingly high: up to 12.7%, with average costs ranging from 2.7 to 4.5% on a high performance virtual machine on IA32 architectures." (Note that the indirect costs due to caching might even be higher.)

X10 version 2.0 till 2.2 had an alternative initialization design called *proto* that allowed cyclic immutable structures at the cost of a more complicated design. In OOPSLA'11, Summers and Müller [12] presented an initialization type-system that is almost identical to our proto proposal, but with different terminology: fully initialized objects are termed *committed* (non-proto), and objects under initialization are termed *free* (proto). Whereas in our proto proposal one cannot read uninitialized fields, in Summers' type-system reading uninitialized fields is allowed and it returns an *unclassified* type (and reading non-null fields from an unclassified type is allowed but it may return null). Phrased differently, reading a field is always allowed, but it may return null even for non-null fields. In contrast, X10 and C++ have types that cannot contain null (in C++ only pointers may be null, and X10 has *structs* which are inlinable objects that may not contain null), thus Summers' type-system is not applicable in such languages. Moreover, proto was used in X10 in the past 3 years prior to X10 2.2, and it was made obselete in favor of the hardhat design presented in this paper because proto did not work well in practice. For example, consider an implementation of `LinkedList` that has a non-null field `header`:

```
class LinkedList extends SuperClass {
  final Entry header = new Entry();
  LinkedList(Collection c) { addAll(c); }
  public boolean addAll(Collection c) {
    // this.header is null if SuperClass calls addAll in its constructor.
  }
}
```

During construction we must read from `this.header`, but `this` is still proto and it is illegal to read from a proto object (in Summer's type-system, reading is allowed but it may return null). It is tempting to think that a dataflow algorithm can prove that `header` was already assigned, however the dataflow must be inter-procedural, and it is further complicated by overriding and `this` escaping. For example, if `SuperClass` calls `addAll` in its constructor, then `header` will still have the default value of `null`. The newly proposed hardhat design can modularly type-check this code assuming that `addAll` is annotated as non-escaping (see Sec. 2).

The *contributions of this paper* are: (i) a complete and strict hardhat design in a full-blown advanced OO language with many cross-cutting concerns (default values, final fields, dataflow analysis, overriding, and especially the concurrent and distributed aspects), (ii) an inter-procedural fixed-point algorithm for definite-async assignment, (iii) implementation inside the X10 open-source compiler and converting the entire X10 code-base (+200K lines of code) to conform to the hardhat principles, (iv) FX10

formalism which is the first to present a flow-sensitive effect system with concurrency constructs and a soundness theorem stating that one can never read an uninitialized field in a statically correct program.

For object initialization rules, the details matter. Instead of basing our work on abstract theoretical discussions, we have chosen to work with a concrete language (X10, see Sec. 3) in which all these rules have been worked out to illustrate the subtleties involved. Our analysis and design will be applicable to any OO language with fine-grained concurrency. Object initialization rules must be dealt with in order to support determinate computation. For example, *Deterministic Parallel Java* (DPJ) [1] also have similar rules for object initialization to prevent `this` from leaking: "the DPJ type and effect system ensures that no other task can access `this` until after the constructor returns."

The remainder of this introduction presents common initialization pitfalls (in sequential, concurrent, and distributed code in both Java and X10) and how the hardhat design prevents them. Specifically, it presents initialization pitfalls in sequential code (Sec. 1.1), concurrent code (Sec. 1.3), and distributed X10 code (Sec. 1.4), and the crux of the hardhat design that prevents these sequential pitfalls (Sec. 1.2).

1.1 Initialization Pitfalls in Sequential Code

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

Executing `new B()` 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 change the type of `b` from `int` to `Integer`, then this code will throw a `NullPointerException`, which is more severe.

The second pitfall is leaking `this` before the object is fully-initialized, for example, `S.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 `S` 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. 1.3 further explains final fields in Java and the implicit synchronization barrier.

1.2 The Crux of the Hardhat Design

The hardhat design in X10 (described in Sec. 2) prevents both pitfalls, because its rules allow dynamic dispatching only when `this` cannot be accessed (first pitfall) and prohibit

```
class A {
 static HashSet S = new HashSet();
 final int a;
 A() {
  a = initA(); // dynamic dispatch!
  System.out.println(toString());
  S.add(this); // leakage!
 }
 int initA() { return 1; }
 public String toString() {
  return "a="+a; }




}
class B extends A {
 int b = 2;
 int initA() { return b+42; }
 public String toString() {
  return super.toString()+",b="+b;}




}
```

(a) Initialization pitfalls

```
class A {
 static HashSet S = new HashSet();
 final int a;
 protected A() {
  a = initA(); // ok
  System.out.println(toStringOfA());
  // S.add(this); // Would be an error
 }
 @NoThisAccess int initA() {return 1;}
 public String toString() {
  return toStringOfA(); }
 @NonEscaping final String toStringOfA(){
  return "a="+a; }
 public static A createA() {
  A res = new A(); S.add(res);
  return res;
 }
}
class B extends A {
 int b = 2;
 @NoThisAccess int initA() {return 42;}
 public String toString() {
  return super.toString()+",b="+b;}
 public static B createB() {
  B res = new B(); S.add(res);
  return res;
 }
}
```

(b) Fixed to conform to the hardhat design

Fig. 1. Two initialization pitfalls in Java: leaking `this` and dynamic dispatch

leaking `this` (second pitfall). We use two method annotations to mark that a method is non-escaping: `@NonEscaping` and `@NoThisAccess`; the first prohibits leaking `this`, and the second is even more strict and prohibits any access of `this`. The essence of the hardhat design are these two rules: (i) Constructors and non-escaping methods may leak/alias `this` *only* to other non-escaping methods (i.e., `this` can only be used as the receiver of a non-escaping method call), (ii) Non-escaping methods are either private or final (thus they cannot be overridden), except `@NoThisAccess` methods that may be overridden but they cannot access `this`. These two rules prevent the two pitfalls of leaking `this` and dynamic dispatching.

Initialization in X10 has the following main attributes: (i) `this` is the only accessible raw/uninitialized object in scope, (ii) only `@NoThisAccess` methods can be dynamically dispatched during construction, (iii) one can read a field *only* after it was assigned, and all fields are assigned by the time the constructor finishes, (iv) reading a final field always results in the same value. (In contrast to Java and [12] where reading a final field might return different values at different times.) Furthermore, with the hardhat rules there is even no need to *zero-initialize all fields* before executing the constructor (as done in Java), thus reducing the program runtime. (We are now in the process of measuring this reduction in runtime; Using a simple bytecode verifier it is possible to ensure that this optimization is safe.)

Fig. 1b shows how to convert the code of Fig. 1a to the hardhat design and thus avoid these two pitfalls (but the original program behavior is changed). We made the following changes: (i) `toString` now delegates to a final non-escaping method `toStringOfA`, and the constructor of `A` can call `toStringOfA`; `B` cannot override this method because it is final, (ii) `initA` is `@NoThisAccess` and therefore `B.initA` cannot read the field `b` (which has not been assigned yet), (iii) instead of leaking `this` into `S` in the constructor of `A`, we refactored the code into two factory methods that create instances of `A` and `B`, and only then add the fully-initialized instance to `S`.

1.3 Initialization Pitfalls in Concurrent Code

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 [9] 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. The memory model does not guarantee *sequential consistency*, but only *weak consistency*. (The barrier would not be needed with sequential consistency.) 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. 1a because `this` was leaked into `S` before the barrier was executed. Consider another thread that iterates over `S` 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.

Java's documentation recommends using final fields when creating an immutable class, and avoid leaking `this` in the constructor. However, `javac` does not even give a warning if that recommendation is violated. To summarize, final fields in Java enable thread-safe immutable objects, but the user must be careful to avoid the pitfall of leaking `this`. The hardhat design in X10 prevents any leakage of `this`, thus making it safe and easy to create immutable classes.

1.4 Initialization Pitfalls in Distributed X10 Code

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 complete 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 that it *globally terminated* when it locally terminated and any activity spawned by `S` also globally terminated. Statement `finish S` blocks until all the activities created by `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 evaluates `E` in place `p`, and finally copies the result back to the original place. Note that `at` is a synchronous construct, meaning that 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).

Fig. 2a (and Fig. 2b) shows how to (correctly) calculate the Fibonacci number `fib(n)` in X10 using concurrent and distributed code. The keywords `val` and `var` are modifiers that correspond to final and non-final variables, respectively. 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 chose the latter because this paper focuses on *object* initialization. X10 has similar initialization rules for final locals and final 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 distributed pitfall, where one assigns to a field of 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 second 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. 2.11).

The hardhat design in X10 prevents both pitfalls by ensuring that all fields of an object are definitely-synchronously assigned when construction of that object ends, and that only fully initialized objects can cross places.

```
class Fib {
 val fib2:Int, fib1:Int, fib:Int;
 def this(n:Int) {

  async {
   val p = here.next();
   at(p) if (n<=1)
    fib2 = 0; else // Err1
    fib2 = new Fib(n-2).fib; // Err1
  }

  if (n<=0)
   fib1 = 0;
  else if (n<=1)
   fib1 = 1;
  else
   fib1 = new Fib(n-1).fib;
  fib = fib2+fib1; // Err2
 }
}
```

(a) Initialization pitfalls in X10

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

(b) Fixed to conform to the hardhat design

Fig. 2. Concurrent and distributed Fibonacci example in X10. 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 among *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`. The two initialization pitfalls: (1) write to field `this.fib2` in another place, which causes (an uninitialized) `this` to be copied to `p`, so one writes to a *copy* of `this` (and the original object is never fully initialized!), (2) read from `fib2` before its write definitely *finished*.

The rest of this paper is organized as follows. Sec. 2 presents the hardhat initialization rules of X10 version 2.2 using examples, by slowly adding language features and describing their interaction with object initialization. Sec. 3 outlines our implementation within the X10 compiler using the polyglot framework, the compilation time overhead of checking these initialization rules, and the annotation overhead in our X10 code base. Sec. 4 presents Featherweight X10 (FX10), which is a formalization of core X10 that includes `finish`, `async`, and flow-sensitive type-checking rules. Sec. 5 summarizes previous work in the field of object initialization. Finally, Sec. 6 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 dispatch, 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, followed by the parallel features of X10 for

writing concurrent code (`finish` and `async`), and distributed code (`at`). Finally, we describe the inter-procedural data-flow analysis that ensures that a field is read only after it has been assigned.

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, all errors issued by the X10 compiler are marked with `//err` (and if there is no such mark then the code is correct).

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()`.

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 explicitly marking non-escaping 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 dispatch may transfer control to the subclass before the superclass completed its initialization. Fig. 3b demonstrates why dynamic dispatch is error-prone during construction: calling `m1` in `A` would dynamically dispatch to the implementation in `B` that would read the default value.

```
class A {
 val a:Int;
 def this() {
  LeakIt.foo(this); //err
  this.a = 1;
  val me = this; //err
  LeakIt.foo(me);
  // so m2 is implicitly non-escaping
  this.m2();
 }
 // permitted to escape
 final def m1() {
  LeakIt.foo(this);
 }
 // implicitly non-escaping
 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();
 }
}
```

(a) Escaping `this` example

```
abstract class C {
 val a1:Int, a2:Int;
 def this() {
  // Can only call non-escaping methods
  this.a1 = m1(); //err1
  this.a2 = m2();
  m4(); m5();
 }
 abstract def m1():Int;
 @NoThisAccess abstract def m2():Int;
 @NonEscaping def m3():void {} // err
 @NonEscaping final def m4():void {}
 @NonEscaping private def m5():void {}
}
class D extends C {
 var b:Int = 3; // non-final field
 def m1() {
  val x = super.a1;
  val y = this.b;
  return 1;
 }
 @NoThisAccess def m2() {
  // Cannot use this or super
  val x = super.a1; //err2
  val y = this.b; //err3
  return 2;
 }
}
```

(b) Dynamic dispatch example

Fig. 3. 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. **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).

X10 prevents dynamic dispatch 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 dispatch 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 must be marked with `@NoThisAccess`. For example, *`err2`* and *`err3`* are caused by rule 4 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 a crash trying to call 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 values of `a1` 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 this or super constructor call. This is one of the reasons why a constructor call must be the first statement in Java; failure to verify this led to a famous security attack [2].

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

Fig. 4. 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 .

Fig. 4 shows that it is an error to try to recover from an exception thrown by a constructor call; the reason 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. 5a 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 create an alias of `this`, 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 can 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.

```
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 {}
}
```

(a) Inner class example: the outer instance is always cooked.

```
class DefaultValuesExample {
  val i0:Int; //err
  // Note the fields below are non-final
  var i1:Int; //ok, has default
  // no default
  var i2:Int{self!=0}; //err
  // ok, has initializer
  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
}
```

(b) Default value example.

Fig. 5. Rule 6: a raw `this` cannot be the receiver of `new`.
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 given a zero initializer..

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

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 one can plug in 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 principle, and a non-final field is implicitly given a zero initializer *if its type has-zero*. Fig. 5b 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 syntactic and semantic, as seen in class `E` of Fig. 6. Syntactically, properties are defined after the class name, must have a type and cannot have an initializer, and 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 E(a:Int) {
  def this(x:Int) {
       property(x);
  }
}
```

```
class F(b:Int) {b==a} extends E {
  val f1 = a+b, f2:Int, f3:E{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 (must be after property(x))
   property(x);
   f3 = new E(this.a);
  }
}
```

Fig. 6. 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) 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. 6: `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 `E` 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 superclass'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. 6) 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 refers to a variable, we say that the variable 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. 7a 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

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


}
```

(a) Closures example.

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

(b) `haszero` type predicate example.

Fig. 7. Rule 11: A closure cannot capture a raw `this`.
Rule 12: A type must be consistent, i.e., it cannot contradict method guards or class invariants.

raw `this`, which 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`.

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 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, a `Box[T]` has a single field of type `T`, and instances of `Box[Byte]` and `Box[Double]` 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. 5b), e.g., if the user instantiates `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 evaluates to true if type `X` has-zero. Using `haszero` in a constraint (e.g., in a class invariant or a method guard), makes it possible to guarantee that a type-parameter will be instantiated with a type that has-zero.

Fig. 7b 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. 5b, a struct has-zero if all its fields has-zero, therefore `WithZeroValue haszero` is true, but `NoZeroValue 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).

We now turn our attention to the parallel features of X10 for concurrent programming (`finish` and `async`) and distributed programming (`at`). Sec. 1.4 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

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

(a) Concurrency in initialization example: asynchronously assign to a field.

```
class A {
 val f:Int;

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

(b) Distributed initialization example.

Fig. 8. Rule 13: A constructor must finish assigning to all fields at least once. **Rule 14:** A final field can be assigned at most once.
Rule 15: a raw `this` cannot be captured by an `at`.

Fig. 8a 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 at the end of the constructor. Assigning to `f3` has an enclosing `finish`, so it is definitely assigned. Field `f1` is also definitely assigned, because it is non-final so from rule 7 it has an implicit zero field initializer. However, field `f2` is final so it does not have an implicit field initializer. Moreover, `f2` is only asynchronously assigned, and the constructor does not have to wait for this assignment to finish, thus violating rule 13. (The exact data-flow analysis to enforce rule 13 is described in Sec. 2.11.) 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. 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 the `CustomSerialization` interface, and defining the method `serialize():SerialData` and the constructor `this(data:SerialData)`.

Fig. 8b 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 therefore rule 15 prohibits `this` to be captured by an `at`. Consequently, we also report that field `f` was not definitely assigned.

2.11 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* (and constructors) 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. 9) by comparison with Java; the full analysis is described in X10's language specification.

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

```
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;
 }

}
```

(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;
 }
}
```

(b)

Fig. 9. 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 assigns, (iii) fields it assigns asynchronously. The data-flow maintains these three sets before and after each statement; *`assigned`* becomes *`asyncAssigned`* after an `async`, and *`asyncAssigned`* becomes *`assigned`* after a `finish`. In this example, we omitted empty sets. **Rule 16:** A field may be read only if it is definitely-assigned. **Rule 17:** A final field may be written only if it is definitely-unassigned.

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 method reads an uninitialized field, then we add it to its *reads* set; however, if a constructor reads an uninitialized field, then it is an error. 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.

3 Implementation

This section discusses our implementation inside the X10 compiler of the hardhat initialization rules. Our X10 code-base of more than 200K lines of code (loc) uses only 104 annotations. We give some measurements such as compilation time and annotation overhead, and conclude with two examples for `@NonEscaping` and `@NoThisAccess`.

The X10 compiler is based on the Polyglot extensible compiler framework, which includes a dataflow framework that has 1309 loc. X10 initialization rules extend this dataflow framework using two classes: one for checking definite-initialization for *local variables* (805 loc), and another for *fields* (951 loc). (The rules of local variables are simpler than those for fields because local variables do not span multiple methods and they must be assigned before use. The focus point of this paper has been object initialization, therefore these rules were not described in this paper.) The dataflow algorithm tracks for each field (or local) the flow of this information: (i) whether the field was read (to find the set of fields each non-escaping method reads), (ii) the minimal and maximal number of times it was sequential and asynchronously written (to make sure a variable is assigned before read, and that final variables are assigned exactly once). The number of times a variable is assigned is sufficient to range between 0, 1, and *more-than-one*, because the error message is the same whether a final variable was assigned twice or more. When flowing out of an `async`, sequential writes become asynchronous writes, and the opposite happens for a `finish`.

Our code-base consists of 5 major components: (i) XRX: X10 runtime and libraries, (ii) SPECjbb: SPECjbb from 2005 converted to X10, (iii) M3R: map-reduce in X10, (iv) UTS: global load balancing library, (v) MISC: those include examples from our programmer guide, our test suite, jira issues, and samples. SPECjbb and M3R are still under development and not publicly available, whereas the rest are open-source and available at `x10-lang.org` (see revision 23028 of `https://x10.svn.sf.net/svnroot/x10/trunk`).

Tab. 1 shows the compilation times broken down according to the time spent for checking fields and locals. We can see that the initialization rules take only a small fraction (0-2%) from the total compilation time, and a maximum of 3.3 seconds for the entire M3R project.

Table 1. Compilation times in milliseconds of our code-base broken down into the time spent by the initialization rules for fields and locals. We used a standard lenovo T500 laptop with 4GB of RAM and Intel Core 2 Duo processor.

	XRX	SPECjbb	M3R	UTS	MISC
Total compilation time	65,241	78,952	254,020	72,205	548,547
Time of checks for fields	156	1,649	3,330	1,272	2,862
Time of checks for locals	32	51	117	33	126

Tab. 2 shows the annotation burden in our code-base. X10 has only two possible method annotations: `@NonEscaping` and `@NoThisAccess`. Recall that all methods transitively called from a constructor are implicitly non-escaping, i.e., the user does not have to explicitly annotate them as `@NonEscaping`, however the compiler issues a warning recommending that they should be marked as such to show intent. Obviously, the number of non-escaping methods is always greater or equal to the number of `@NonEscaping` annotations. As can be seen, the annotations burden is minor: only 104 annotations in total.

Table 2. The annotation burden in our code-base

	XRX	SPECjbb	M3R	UTS	MISC
# of lines	27,153	14,603	71,682	2,765	155,345
# of files	257	63	294	14	2,283
# of constructors	276	267	401	23	1,297
# of methods	2,216	2,475	2,831	124	8,273
# of non-escaping methods	8	38	34	3	83
# of `@NonEscaping`	7	7	13	1	62
# of `@NoThisAccess`	1	0	1	0	12

Our applications only use `@NoThisAccess` twice: once in M3R to allow a subclass to determine the value of a final field of the superclass during initialization, and the second time in XRX in method `typeName()` of interface `Any` (this method may be overridden and it is often called during construction for debugging purposes).

The following example shows a common pattern for using `@NonEscaping` and a common refactoring that was done when converting Java code to X10. Class `HashMap` in Java calls `put` in two constructors: the deserialization constructor and the copy constructor (that gets a map argument and creates a copy of that map). However, `put` is not a final method and it might be useful to override it in subclasses, and therefore it cannot be called during construction. Thus, we refactored this code in X10 and called instead a non-escaping method called `putInternal` and method `put` delegates to that method:

```
public def put(k: K, v: V) { putInternal(k,v); }
@NonEscaping protected final def putInternal(k:K, v:V) { ... }
```

A similar refactoring was also done in `HashMap` for method `rehash`.

Asynchronous initialization was not used in our big applications because they predate this feature. (It is used in our smaller examples and tests more than 50 times.)

This pattern is especially useful for local variables, and more importantly, the analysis prevents bugs such as:

```
val x:Int; val y:Int;
finish { async { x = doCalculation1(); }
  y = doCalculation2(); // WRONG to use variable x here
} // OK to use variable x now
```

4 Formalism: FX10

Featherweight X10 (FX10) is a formal calculus for X10 intended to complement Featherweight Java (FJ). It models imperative aspects of X10 including the concurrency constructs `finish` and `async`. FX10 models the heart of the field initialization problem: a field can be read only after it is definitely assigned.

The basic idea behind the formalization is very straightforward. We break up the formalization into two distinct but interacting subsystems, a *type system* (Sec. 4.2) and an *effect* system (Sec. 4.3). The type system is completely standard – think the system of FJ, adapted to the richer constructs of FX10.

The effect system is built on a very simple *logic of initialization assertions*. The primitive formula $+\mathtt{x}$ ($+\mathtt{p.f}$) asserts that the variable `x` (the field `f` of `p`) is definitely initialized with a cooked object, and the formula $-\mathtt{x}$ ($-\mathtt{p.f}$) asserts that it is being initialized by a concurrent activity (and hence it will be definitely initialized once an enclosing `finish` is crossed). An *initialization formula* ϕ or ψ is simply a conjunction of such formulas $\phi \wedge \psi$. An *effects assertion* ϕ `S` ψ (for a statement `S`) is read as a partial correctness assertion: when executed in a heap H that satisfies the constraint ϕ, `S` will on termination result in a heap H' that satisfies ψ. Since we do not model `null`, our formalization can be particularly simple: variables, once initialized, stay initialized, hence H' will also satisfy ϕ (see Sec. 4.4 for a definition of heaps and when a heap satisfy ϕ).

Another feature of our approach is that, unlike Masked Types [10], the source program syntax does not permit the specification of initialization assertions. Instead we use a standard least fixed point computation to automatically decorate each method $\mathtt{def\ m}(\overline{\mathtt{x}} : \overline{\mathtt{C}})\{\mathtt{S}\}$ with pairs (ϕ, ψ) (in the free variables $\mathtt{this}, \overline{\mathtt{x}}$) such that under the assumption that all methods satisfy their corresponding assertion we can show that ϕ `S` ψ.[1] This computation must be sensitive to the semantics of method overriding, that is a method with decoration (ϕ, ψ) can only be overridden by a method with decoration (ϕ', ψ') that is "at least as strong as" (ϕ, ψ) (viz, it must be the case that ($\phi \vdash \phi'$ and $\psi' \vdash \psi$). Further, if the method is not marked `@NonEscaping`, then ϕ is required to entail $+\mathtt{this}$ (that is, `this` is cooked), and if it marked `@NoThisAccess` then ϕ, ψ cannot have `this` free.

[1] Note that this approach permits a formal `x` to a method to be completely raw (ϕ does not entail $+\mathtt{x}$ or $+\mathtt{x.f}$ for any field `f`) or partially raw (ϕ does not entail $+\mathtt{x}$ but may entail $+\mathtt{x.f}$ for some fields `f`). As a result of the method invocation the formal may become more cooked. In X10, in order for the inference to be intra-class, we require that all method parameters $\overline{\mathtt{x}}$ (except `this`) are cooked, i.e., $+\mathtt{x}_i$. In FX10 we are more relaxed and allow methods to receive and initialize raw parameters.

By not permitting the user to specify initialization assertions we make the source language much simpler than [10] and usable by most programmers. The down side is that some initialization idioms, such as cyclic initialization, are not expressible.

For reasons of space we do not include the details behind the decoration of methods with initialization assertions. We also omit many extensions (such as generics, interfaces, constraints, casting, inner classes, overloading, co-variant return types, final, field initializers etc.) that were discussed in the first half of the paper. FX10 also does not model places because the language design decision to only permit cooked objects to cross places means that the rules for `at` are routine.

We use the usual notation of $\overline{\mathtt{x}}$ to represent a vector or set of $\mathtt{x}_1, \ldots, \mathtt{x}_n$. A program `P` is a pair of class declarations $\overline{\mathtt{L}}$ (that is assumed to be global information) and a statement `S`.

4.1 Syntax

Fig. 10 shows the syntax of FX10. Expression `val x = e;S` evaluates `e`, assigns it to a new variable `x`, and then evaluates `S`. The scope of `x` is `S`.

The syntax is similar to the real X10 syntax with the following difference: FX10 does not have constructors; instead, an object is initialized by assigning to its fields or by calling non-escaping methods.

$\mathtt{P} ::= \overline{\mathtt{L}}, \mathtt{S}$	Program.
$\mathtt{L} ::= \mathtt{class\ C\ extends\ D\ \{}\ \overline{\mathtt{F}}; \overline{\mathtt{M}}\ \mathtt{\}}$	cLass declaration.
$\mathtt{F} ::= \mathtt{var\ f : C}$	Field declaration.
$\mathtt{M} ::= \mathtt{G\ def\ m}(\overline{\mathtt{x}} : \overline{\mathtt{C}}) : \mathtt{C}\{\mathtt{S}\}$	Method declaration.
$\mathtt{G} ::= \mathtt{@NonEscaping} \mid \mathtt{@NoThisAccess}$	Method modifier.
$\mathtt{p} ::= \mathtt{l} \mid \mathtt{x}$	Path.
$\mathtt{e} ::= \mathtt{p.f} \mid \mathtt{new\ C}$	Expressions.
$\mathtt{S} ::= \mathtt{p.f} = \mathtt{p}; \mid \mathtt{p.m}(\overline{\mathtt{p}}); \mid \mathtt{val\ x = e;S}$ $\mid \mathtt{finish\ \{S\}} \mid \mathtt{async\ \{S\}} \mid \mathtt{S\ S}$	Statements.

Fig. 10. FX10 Syntax. The terminals are locations (`l`), parameters and `this` (`x`), field name (`f`), method name (`m`), class name (`B`,`C`,`D`,`Object`), and keywords (`new`, `finish`, `async`, `val`). The program source code cannot contain locations (`l`), because locations are only created during execution/reduction in R-NEW of Fig. 12.

4.2 Type System

The type system for FX10 checks that every parameter and variable has a type (a type is the name of a class), and that a variable of type `C` can be assigned only expressions whose type is a subclass of `C`, and can only be the receiver of invocations of methods defined in `C`. The type system is formalized along the lines of FJ. No complications are introduced by the extra features of FX10 – assignable fields, local variable declarations, `finish` and `async`. We omit details for lack of space and because they are completely routine. In the rest of this section we shall assume that the program being considered $\overline{\mathtt{L}}, \mathtt{S}$ is well-typed.

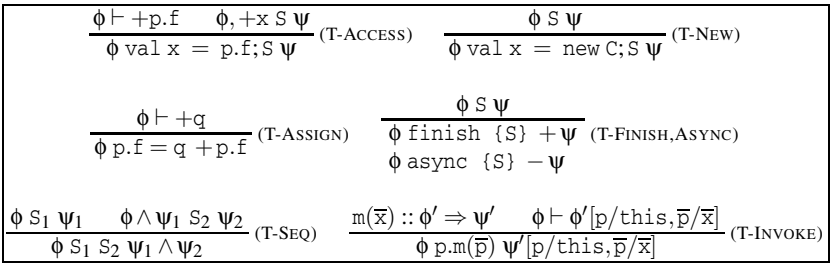

Fig. 11. FX10 Effect System (ϕ S ψ)

4.3 Effect System

We use a simple logic of initialization for our basic assertions. This is an intuitionistic logic over the primitive formulas $+\mathtt{p}$ (the variable or parameter p is initialized), $+\mathtt{p.f}$ (the field p.f is initialized), and $-\mathtt{p}, -\mathtt{p.f}$ (it is being *concurrently* initialized). We are only concerned with conjunctions and existential quantifications over these formulas: $\phi, \psi ::= \mathtt{true} \mid +\mathtt{x} \mid +\mathtt{p.f} \mid -\mathtt{p.f} \mid \phi \wedge \psi$

The notion of substitution on formulas $\phi[\overline{\mathtt{x}}/\overline{\mathtt{z}}]$ is specified in a standard fashion.

The inference relation is the usual intuitionistic implication over these formulas, and the following additional proof rules: (1) if $\phi \vdash +\mathtt{p}$ then $\phi \vdash -\mathtt{p}$; (2) if $\phi \vdash +\mathtt{p.f}$ then $\phi \vdash -\mathtt{p.f}$; (3) if $\phi \vdash +\mathtt{p}$ ($\phi \vdash -\mathtt{p}$) then $\phi \vdash +\mathtt{p.f}$ ($\phi \vdash -\mathtt{p.f}$); and (4) if the *exact* class of p is C, and C has the fields $\overline{\mathtt{f}}$, then $\phi \vdash +\mathtt{p}$ ($\phi \vdash -\mathtt{p}$) if $\phi \vdash +\mathtt{p.f}_i$ ($\phi \vdash -\mathtt{p.f}_i$), for each *i*. (We only know the *exact* class for a local p when $\mathtt{val\ p\ =\ new\ C;S}$.)

The proof rules for the judgement ϕ S ψ are given in Figure 11. They use two syntactic operations on initialization formulas defined as follows. $+\psi$ is defined inductively as follows: $+\mathtt{true} = \mathtt{true}$, $+\pm\mathtt{x} = +\mathtt{x}$, $+\pm\mathtt{p.f} = +\mathtt{p.f}$, $+(\phi \wedge \psi) = (+\phi) \wedge (+\psi)$. $-\psi$ is defined similarly: $-\mathtt{true} = \mathtt{true}$, $-\pm\mathtt{x} = -\mathtt{x}$, $-\pm\mathtt{p.f} = -\mathtt{p.f}$, $-(\phi \wedge \psi) = (-\phi) \wedge (-\psi)$.

The rule (T-ACCESS) can be read as asserting: if ϕ entails the field p.f is initialized (together with $+\mathtt{x}$ which states that x is initialized to a cooked object), we can establish that execution of S satisfies the assertion ψ then we can establish that execution of $\mathtt{val\ x\ =\ p.f;S}$ in (a heap satisfying) ϕ establishes ψ. Here we must take care to project x out of ψ since x is not accessible outside its scope S; similarly we must take care to project x out of ϕ when checking S. The rule (T-NEW) can be read in a similar way except that when executing S we can make no assumption that x is initialized, since it has been initialized with a raw object (none of its fields are initialized). Subsequent assignments to the fields of x will introduce effects recording that those fields have been initialized. The rule (T-ASSIGN) checks that q is initialized to a cooked object and then asserts that p.f is initialized to a cooked object. The rule (T-FINISH) can be understood as recording that after a finish has been "crossed" all asynchronous initializations ψ can be considered to have been performed ϕ. Conversely, the rule (T-ASYNC) states that any initializations must be considered asynchronous to the surrounding context. The rule (T-SEQ) is a slight variation of the stadard rule for sequential composition that permits ϕ to be used in the antecedent of $\mathtt{S}_2$, exploiting monotonicity of effects. Note the effects recorded for $\mathtt{S}_1\ \mathtt{S}_2$ are a conjunction of the effects recorded for $\mathtt{S}_1$ and $\mathtt{S}_2$. The rule (T-INVOKE) is routine.

As an example, consider the following classes. Assertions are provided in-line.

```
class A extends Object {
  var f:Object; var g:Object; var h:Object;
  @NonEscaping def build(a:Object) {
    // inferred decoration: phi => psi
    //   phi= +this.g, +a
    //   psi= -this.h, +this.f
    // checks phi implies +this.g
    val x = this.g;
    async { this.h = x; } // psi= -this.h
    finish {
      // checks phi implies +a
      async { this.f = a; } // psi= -this.h,-this.f
    } // psi= -this.h,+this.f
  }
}
class B extends A { e:Object; }
```

Method `build` synchronously (asynchronously) initializes fields `this.f` (`this.h`), and it assumes that `this.g` and `a` are cooked. The following statement completely initializes `b`:

```
val b = new B();
val a = new Object(); // psi= +a
b.g = a; // psi= +a,+b.g
finish {
  b.build(a); // psi= +a,+b.g,+b.f,-b.h
  async { b.e = a; } // psi= +a,+b.g,+b.f,-b.h,-b,-b.e,-b
} // psi= +a,+b
```

4.4 Reduction

A *heap* H is a mapping from a given set of locations to *objects*. An object is a pair $C(F)$ where C is a class (the exact class of the object), and F is a partial map from the fields of C to locations. We say the object `l` is *total/cooked* (written $cooked_H(\texttt{l})$) if its map is total, i.e., $H(\texttt{l}) = \texttt{C}(F) \qquad \mathrm{dom}(F) = \mathit{fields}(\texttt{C})$.

We say that a heap H *satisfies* ϕ (written $H \vdash \phi$) if the plus assertions in ϕ (ignoring the minus assertions) are true in H, i.e., if $\phi \vdash +\texttt{l}$ then `l` is cooked in H and if $\phi \vdash +\texttt{l.f}$ then $H(\texttt{l}) = \texttt{C}(F)$ and $F(\texttt{f})$ is cooked in H.

The reduction relation is described in Figure 12. An S-configuration is of the form $\texttt{S}, H$ where `S` is a statement and H is a heap (representing a computation which is to execute `S` in the heap H), or H (representing terminated computation). An E-configuration is of the form $\texttt{e}, H$ and represents the computation which is to evaluate `e` in the configuration H. The set of *values* is the set of locations; hence E-configurations of the form $\texttt{l}, H$ are terminal.

Two transition relations $\rightsquigarrow$ are defined, one over S-configurations and the other over E-configurations. For X a partial function, we use the notation $X[v \mapsto e]$ to represent the partial function which is the same as X except that it maps v to e. The rules defining these relations are standard. The only minor novelty is in how `async` is defined. The critical rule is the last rule in (R-STEP) – it specifies the "asynchronous" nature of `async`

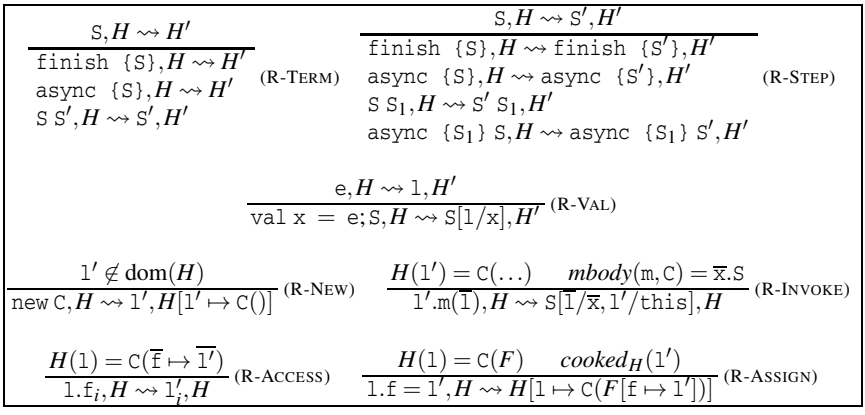

Fig. 12. FX10 Reduction Rules ($\mathtt{S},H \rightsquigarrow \mathtt{S}',H' \mid H'$ and $\mathtt{e},H \rightsquigarrow \mathtt{l},H'$)

by permitting S to make a step even if it is preceded by async {S_1}. The rule (R-NEW) returns a new location that is bound to a new object that is an instance of C with none of its fields initialized. The rule (R-ACCESS) ensures that the field is initialized before it is read (f_i is contained in $\overline{f}$).

4.5 Results

We say a heap H is *correctly cooked* (written $\vdash H$) if a field can point only to cooked objects, i.e., for every object $o = C(F)$ in the range of H and every field $f \in \mathrm{dom}(F)$ it is the case that every object $l = H(F(f))$ is cooked ($cooked_H(l)$). We shall only consider correctly cooked heaps (valid programs will only produce correctly cooked heaps). As the program is executed, the heap monotonically becomes more and more cooked. Formally, H' is *more cooked* than H (written $H' \vdash H$) if for every $l \in \mathrm{dom}(H)$, we have $H(l) = C(F)$, $H'(l) = C(F')$, and $\mathrm{dom}(F) \subseteq \mathrm{dom}(F')$.

A *heap typing* Γ is a mapping from locations to classes. H is said to be typed by Γ if for each $l \in \mathrm{dom}(H)$, the class of $H(l)$ is a subclass of $\Gamma(l)$. Since our treatment separates out effects from types, and the treatment of types is standard, we shall assume that all programs and heaps are typed.

A statement S is *closed* (written $\vdash$ S) if it does not contain any free variables. We say that S is *annotatable* if there exists ϕ, ψ such that ϕ S ψ can be established.[2]

We say that a program $P = \overline{L}S$ is *proper* if it is well-typed and each method in L can be decorated with pre-post assertions (ϕ, ψ), and S is annotatable. The decorations must satisfy the property that under the assumption that every method satisfies its assertion (this is for use in recursive calls) we can establish for every method def m($\overline{x} : \overline{C}$){S} with assertion (ϕ, ψ) that it is the case that the free variables of ϕ, ψ are contained in this, $\overline{x}$, and that ϕ S ψ.

[2] An example of a statement that is *not* annotatable is val x = new C;val y = x.f;z.g = y where C has a field f. This attempts to read a field of a variable initialized with a brand-new object.

We prove the following theorems. In all these theorems the background program $\mathtt{P}$ is assumed to be proper. The first theorem is analogous to subject-reduction for typing systems.

Theorem 1. ***Preservation*** *Let* $\phi\ \mathtt{s}\ \psi, \vdash \mathtt{s}, \vdash H, H \vdash \phi$. *(a) If* $\mathtt{s}, H \rightsquigarrow H'$ *then* $\vdash H', H' \vdash H, H' \vdash +\psi$. *(b) If* $\mathtt{s}, H \rightsquigarrow \mathtt{s}', H'$ *then* $\vdash \mathtt{s}', \vdash H', H' \vdash H$, *there exists* ϕ', ψ' *such that* $H' \vdash \phi', \phi'\ \mathtt{s}'\ \psi', \phi' \vdash \phi, \psi' \vdash \psi$.

Theorem 2. ***Progress*** *Let* $\phi\ \mathtt{s}\ \psi, \vdash \mathtt{s}, \vdash H, H \vdash \phi$. *The configuration* $\mathtt{s}, H$ *is not stuck.*

For proofs, please see associated technical report.

Because our reduction rules only allow reads from initialized fields, a corollary is that a field can only be read after it was assigned, and an attempt to read a field will always succeed.

5 Related Work

A static analysis [11], 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 [6] 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 [7].

An early work to support non-null types in Java [3] 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. Our approach permits optimization 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 [4,10] 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 [10] present types that describe the exact fields that have not yet been initialized. Summers and Müller [12] describe a simpler type system that is almost identical to our `proto` design, however they only treat non-null types and they allow reading a field before it was assigned. Our type system is simpler but less expressive because it cannot handle immutable cyclic structures.

There is also a time-aware type system [8] 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 synchronization). The same mechanisms can also be used to express when an object becomes cooked.

Ownership types can be used to create immutable cycles [14]. 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. Our approach does not use ownership types.

6 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 were incorporated in the open-source X10 compiler, and are being used in production code.

References

1. Bocchino Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel java. In: OOPSLA 2009, pp. 97–116. ACM, New York (2009)
2. Dean, D., Felten, E., Wallach, D.S.: Java security: From hotjava to netscape and beyond. In: IEEE Symposium on Security and Privacy, pp. 190–200 (1996)
3. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: OOPSLA 2003, pp. 302–312 (2003)
4. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA 2007, pp. 337–350 (2007)
5. Gil, J., Itai, A.: The Complexity of Type Analysis of Object Oriented Programs. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 601–634. Springer, Heidelberg (1998)
6. Gil, J.Y., Shragai, T.: Are We Ready for a Safer Construction Environment? In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 495–519. Springer, Heidelberg (2009)
7. Hubert, L., Jensen, T., Monfort, V., Pichardie, D.: Enforcing Secure Object Initialization in Java. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 101–115. Springer, Heidelberg (2010)
8. Matsakis, N.D., Gross, T.R.: A time-aware type system for data-race protection and guaranteed initialization. In: OOPSLA 2010, pp. 634–651 (2010)
9. Pugh, W.: JSR 133: Java memory model and thread specification revision (2004), http://jcp.org/en/jsr/detail?id=133
10. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL 2009, pp. 53–65 (2009)
11. Seo, S., Kim, Y., Kang, H.-G., Han, T.: A static bug detector for uninitialized field references in java programs. IEICE - Trans. Inf. Syst. E90-D, 1663–1671 (2007)
12. Summers, A.J., Müller, P.: Freedom before commitment - a lightweight type system for object initialisation. In: OOPSLA 2011 (2011)
13. Yang, X., Blackburn, S.M., Frampton, D., Sartor, J.B., McKinley, K.S.: Why nothing matters: the impact of zeroing. In: OOPSLA 2011, pp. 307–324. ACM, New York (2011)
14. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in generic java. In: OOPSLA 2010, pp. 598–617 (2010)