Object Initialization in X10

Yoav Zibin (**)
David Cunningham (*)
Igor Peshansky (**)
Vijay Saraswat (*)

(*) IBM TJ Watson Research Center
(**) Google Labs (ex IBMers)

ECOOP2012
Avoid access to uninitialized fields

Well-studied problem in Java
X10 is very like Java

What is interesting about X10?
• Uninitialized fields more serious than Java
• Interactions with distribution/concurrency
• Pragmatic balance of power/bureaucracy/simplicity
Initialization Can be Tricky

- Reason 1: dynamic dispatching

```java
abstract class A {
    A() {
        System.out.println("me="+this.<span class="highlighted">description</span>());
    }
    abstract String <span class="highlighted">description</span>();
}
class B extends A {
    int b = 2;
    public String <span class="highlighted">description</span>() { 
        return "b="+b;
    }
}
```

In X10, one could write `Int{self!=0}`
Initialization Can be Tricky

• Reason 2: leaking an uninitialized object

class A {
    public static HashSet set = new HashSet();
    A() {
        set.add(this);
    }
}
class B extends A {
    final int b = 2;
}
Desired Initialization Properties

• Cannot read uninitialized fields
  – C++: uninitialized fields have unspecified value
  – Java: fields are zero initialized

• Final fields
  – Written exactly once
  – Single value for final fields

• Immutable objects are thread-safe

• Type safety in X10’s type system
  – i.e. for Int{self!=0}, Object{self!=null}, etc

• Default initialize if possible
X10 Initialization: Hard hat design

Strict
• limit dynamic dispatching on \texttt{this}
• no leaking \texttt{this}

Pros
• Has desired language properties
• Annotation overhead
• Compile speed
• Simplicity
• Expressive enough (for us)

Cons
• Can’t express cyclic immutable object graphs
Constructor rules (@NonEscaping)

- Flow sensitive analysis:
  - Cannot read uninitialized fields
  - Can assign only once

- **this** cannot escape (assign to var, field, param of method, etc)
- **this.m()** must be final/static/private (with 1 exception...)
- (Body of **m()** also subject to constructor rules)
- If **m()** in a superclass, must be annotated (separate compilation)

```scala
class A {
  @NonEscaping
  final def m3() {}  
}

class B extends A {
  val f:Int;
  def this() {
    m1();
    f = 42;
    LeakIt.leak(this);  // ERR
  }
  private def m1() { m2(); }  
  final def m2() { m3(); }  
}
```
Constructor rules (@NoThisAccess)

Real example: Taken from Ogre3D, a free rendering engine

Parameters also allowed

class Tree {
    val root:Node;
    def this() {
        root = makeNode();
    }
    @NoThisAccess
    def makeNode() = new Node();
}

class MyNode extends Node { }

class MyTree extends Tree {
    @NoThisAccess
    def makeNode() = new MyNode();
}
Default initialization: Has-zero

• Definition of $T$ haszero
  – A type $\text{haszero}$ if it has null, false, or 0
  – Extend to X10 structs recursively

• A \texttt{var} field that lacks an initializer and whose \texttt{type} \text{haszero}, is implicitly zero-initialized.

```java
class A {
  var i0:Int;
  var i1:Int{self!=0}; //ERR
  val i2:Int; //ERR
}
```
Generics

- **haszero** type predicate

```scala
class B[T] {T haszero } {
  var f1:T;
  val f2:T = Zero.get[T]();
}
class Usage {
  var b1:B[Int];
  var b2:B[Int{self!=0}]; //ERR
}
class Array[T] {
  def this(size:Int) {T haszero} {...}
  def this(defaultElement:T,size:Int) {...}
}
```
Concurrent programming

class A {
    val f1:Int;
    val f2:Int;
    def this() {
        async f1 = 2; // ERR
        finish {
            async f2 = 3;
        }
    }
}

• Using scoped language features for sync means compiler can understand it easily
• Rules are strict, but can still parallel-initialize
class A {
    var i:Int{self!=0} , j:Int{self!=0};
    def this() {
        finish {
            asyncWriteI(); // asyncAssigned={i}
            writeJ(); // ERR
        } // assigned={i}
        writeJ(); // assigned={i,j}
    }
    private def asyncWriteI () { // asyncAssigned={i}
        async this.i=1;
    }
    private def writeJ() { // reads={i} assigned={j}
        if (this.i==1) this.j = 2; else this.j = 1;
    }
}
Distributed programming

- **at** shifts execution to another place
- Implies serialization (field access)
- So **at** cannot capture uninitialized **this**.

```scala
class A {
  val f1: Int { self != 0; }
  val f2: Int { self != 0; }
  def this() {
    this.f1 = 1;
    at (here.next()) {
      Console.OUT.println(this.f1); //ERR
    }
  }
}
```
Previous work

- C++: don’t do it
- Java: don’t do it but at least type safe
- Non-null types
- Summers & Muller (similar to old X10)
- Masked types (Qi and Meyers)
- Detector uninitialized fields
Evaluation / Conclusion

• Language is safe (formalism in paper)
• Annotations rarely required (analysis in paper)
• Sufficiently expressive for ~300000 LOC
  – (but can’t do immutable object cycles)
• Simple rules (simpler than related work)
Questions?
Constructor Rules Overview

- Demonstrate the rules by examples
- Initialization is a cross-cutting concern
  - Dynamic dispatching and leaking `this`
  - Default/zero value
  - Generics
  - Concurrent and distributed programming
  - More in the paper:
    - Inner classes, properties, exceptions, closures, structs, serialization, ...
Conclusion

• Java and C++ initialization is error-prone
• X10 initialization design
  – Strict: protects from errors
  – Simple
  – Flexible (but cannot express cyclic immutability)
  – Type safe
  – Final fields has a single value
• See paper for alternative designs (e.g., proto)
• Questions?
Initialization Can be Tricky

• Reason 3: concurrent and distributed code

```scala
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;
  }
}
```
Properties

• Properties are final fields that are initialized first using `property(...);`

```scala
class A(a:Int) {}
class B(b:Int) {b==a} extends A {
  val f1 = a+b;
  def this(x:Int) {
    super(x);
    val i1 = super.a;
    val i2 = this.f1; //ERR
    property(x);
    val i3 = this.f1;
  }
}
```

Field initializers are executed after `property(...)`