Determinate, deadlock-free imperative data-structures

Abstract

Imperative memory locations are famously indeterminate under concurrent read/write operations. This means that it is very difficult to systematically design data-structures that are determinate and deadlock-free.

There are two fundamental ways to ensure determinacy – run-time checks and static types.

The former can lead to run-time overheads and the latter to brittle programs since it is very difficult to smoothly extend the rich type systems underlying modern imperative languages (supporting objects, inheritance, subtyping, genericity etc) to support determinacy. The fundamental problem is the possibility of aliasing in arbitrary unstructured heaps. This makes it very difficult to get a static handle on concurrent access to a shared location.

In this paper we show that a middle path can be successful for a language based on structured concurrency, such as X10. We introduce two new abstractions – *accumulators* and *clocked values* with very modest compiler support. Accumulators of type `T` permit multiple concurrent writes, these are reduced into a single value by a user-specified reduction operator. Clocked values of type `T` operate on two values of `T` (the *current* and the *next*). Read operations are performed on the current value and write operations on the next. Such values are implicitly associated with a clock [**?**] and the current and next values are switched (determinately) on quiescence of the clock. Clocked values capture the common "double buffering" or "red/black" concurrency idiom.

We show that these abstractions (by design) determinate and deadlock-free in any usage, though executions may throw exceptions under certain circumstances. We show that these data-structures are very natural to use and successfully capture many common patterns of expression that are semantically determinate (e.g. histograms, all-to-all reductions, stencil computations etc). We show that there are simple statically-checkable rules that can establish for many common idioms that concurrency-related exceptions will not be thrown at run-time, and that some potentially costly synchronization checks can be avoided.

A key technical innovation is the introduction of an *implicit ownership domain* for objects. This provides a way around the unstructured heap by permitting only the current activity and its spawned asyncs to access the objects. Now the block structure of `finish`, `async`, `at` and the clock construct of X10 can be used to establish determinacy in a local way, independent of the context in which the code is being used.

1. Introduction

Determinate, deadlock-free imperative data-structures

Desiderata:

* Data-structures should by design by dynamically determinate and deadlock-free. * Usable * Static type-checking can provide extra guarantees (e.g. no concurrency related run-time exceptions).

We discuss three examples

* clocks * accumulators * clocked types

Challenge Arbitrary nature of object graphs.

1. *Use activity registration as a mechanism to tame object graphs.*
2. *Focus on structured concurrency. Using scoping and block-structure to delimit regions of code that may execute in parallel and affect the data structure.*
3. *Accumulation can be defined safely by delaying. However, the delay operation is guaranteed to be deadlock-free.*
4. *Clocked types support phased computation, another common idiom particularly for stencil computations.*

Key contributions:

1. *Identification of determinate, deadlock-free data-structures.*
2. *Discussion of design alternatives which points out the difficulty of integrating these ideas in a modern OO language.*
3. *Discussion of various idioms expressible using these data-structures.*
4. *Proof of determinacy and deadlock-freedom in an abstract version of the language.*

These constructs are implemented in X10, available as open source from SVN head and will be in the next release of X10.

Semantics and theorems for an abstract version of the language.

2. X10

Dscussion of X10.

2.0.1 `sync`

We introduce a new derivative synchronization concept, that of syncing.

We introduce `Runtime.sync()`. It returns only after all asyncs spawned by the current activity have terminated or stopped at an `advance` on a clock registered on the current activity.

Programs using finish/async/at/atomic/clocks/Runtime.sync() cannot deadlock.

3. Accumulators and Clocked Types

3.1 Accumulators

Each async (dynamically) has a set of registered `@Sync` accumulators and `@Async` accumulators.

- The registered `@Async` accumulators for an activity are the registered `@Sync` and `@Async` accumulators of its parent activity.
- The registered `@Sync` accumulators for an activity are the ones it has created.

This permits computations to be determinate even though objects can be stored in heaps, since no async other than a child of the async that creates the accumulator can actually operate on them. One can still use the flexibility of the heap to arrange for complex data-dependent transmission pathways for the accumulator from point of creation to point of use. e.g. arrays of accumulators, hash-maps, etc.

The method

```
Runtime.isRegistered[T](x:Acc[T]):Int
```

returns `0` if `x` is not registered with the current activity, `1` if it is `@Sync` registered, and `2` if it is `@Async` registered.

// * Note: Runtime.isRegistered(Clock):Boolean should also be provided // for symmetry.

An invocation `e.m(e1,...,en)` of an `@Sync` method on an `Acc` is translated to:

```
{
  val x = e;
  if (Runtime.isRegistered(x) !=1)
    throw new IllegalAccAccess(x);
  Runtime.sync();
  x.m(e1,...,en)
}
```

The `@Sync` methods on an `Acc` are ones that return its current value (`@Read`) and ones that reset it (`@Write`).

An invocation `e.m(e1,...,en)` of an `@Async` method on an `Acc` is translated to:

```
{
  val x = e;
  if (Runtime.isRegistered(x)==0)
    throw new IllegalAccAccess(x);
  x.m(e1,...,en);
}
```

The only `@Async` method on an `Acc` is the one that offers an update to its value (`@Write`).

In many cases the compiler can statically evaluate whether `Runtime.isRegistered(x) > 0` and/or whether a call to `Runtime.sync()` will suspend.

It may then appropriately simplify the above code.

e.g. in the code below

```
val x:Acc[Int] = new Acc[Int](0, Int.+);
finish for (i in 0..100000) async
   x <- i;
Console.OUT.println("x is " + x());
```

the compiler can infer that `x()` wont suspend, due to the `finish`. Hence it may eliminate the run-time suspension check. Further it can establish that `x` is `@Sync` registered with the current activity, hence it can eliminate the access check.

Notes:

- There are no restrictions to storing Accs in data-structures or reading them.
- However, any attempt to use it will fail unless the Acc is registered with the current activity.
- The runtime checks in Runtime.sync() and Runtime.registered(..) ensure that the operations on an Acc are determinate.
- Static analyses similar to the ones performed for clocks may be performed to ensure that exceptions are not thrown.

Proposition 3.1. *`Acc`'s are determinate under arbitrary usage.*

3.2 Example use of `Acc`

Example 3.2 (Histogram). `@det`

```
def histogram(N:Int,
    A:Rail[Int(0..N)]):Rail[Int](N+1) {
    val result = new Rail[Acc[Int]](N+1, (Int)=>new
    Acc[Int](0,Int.+));
    finish for (i in A.values()) async {
       result(i) <- 1;
    }
    return new Rail[Int](N+1, (i:Int)=> result(i));
}
```

Example 3.3 (Distributed word-count). `// A DistHashMap is used because the input is a DistStream`

```
@det
def
    wordCount(m:DistStream[Word]):DistHashMap[Word,Int](m.dist)
    {
  val a = new DistHashMap[Word, Acc[Int]](m.dist,
          (w:Word)=> new Acc[Int](Int.Sum)));
  finish for (p in m.dist.places()) async at(p) {
     for (word in m(p).words())
          a(word)<- 1;
  }
  return  new DistHashMap[Word, Int](m.dist,
    (w:Word)=> a(w));
}
```

Accumulators can be used to implement collective operations such as all-to-all reductions in a straightforward "shared memory" style.

Here we show the single-sided, blocking version.

Example 3.4. `@det`

```
def reduce[T](in:DistArray[T], red:Reducible[T]):T {
  val acc = new Acc[T](red);
  val temp = new GlobalRef[Acc[T]](acc);
  finish for (dest in in.dist.places()) async
     at(dest) {
     val local = new Acc[T](red);
     for (p in in.dist | here) {
        local <- in(p);
     }
     val x = local();
     async at(origin) temp() <- x;
  }
  return acc();
}
```

An `allReduce` can be implemented by following the above operation with a broadcast:

```
@det
def
   allReduce[T](in:DistArray[T]{self.dist==Dist.UNIQUE},
  red:Reducible[T], out:DistArray[T](in.dist)):void {
  val x = reduce(in, red);
  finish for (dest in out.dist.places()) async at
   (dest) {
    for (p in out.dist |here)
      out(p)=x;
  }
}
```

One can write this code using a clock (to avoid two finish nests).

The collective style requires extending clock so the advance method takes arbitrary args and performs collective operations on them, mimicking the MPI API.

3.3 Clocked types

The central idea behind clocked data-structures is that read/write conflicts are avoided using "double buffering." Two versions of the data-structure are kept, the *current* and the *next* versions. Reads can be performed simultaneously by multiple activities – they are performed on the current version of the data-structure. Writes are performed on the next version of the data-structure. On detection of termination of the current phase – when all involved activities are quiescent – the current and the next versions are switched.

`Clocked[T]` and `ClockedAcc[T]` are distinguished in that unlike the former the latter permits accumulation operations.

Clocked objects are registered with activities, just like accumulators. This permits computations to be determinate even though objects can be stored in heaps, since no async other than a child of the async that creates the clocked object can actually operate on them.

Each async (dynamically) has a set of registered clocked values. The registered clocked values for an activity are the clocked values it has created, and the ones registered to its parent activity.

`Clocked[T]` has a constructor that takes two `T` arguments, these are used to initialize the now and next fields. These arguments should be "new" (that is, no other data-structure should have a reference to these arguments).

For `x:Clocked[T]` the following operations available to any activity on which `x` is registered:

- `x()` – this returns the value of the current field.
- `x() = t` – this sets the value of the next field. Note: write-write conflicts are possible since multiple activities may try to set the value at the same time.
- `x.finalized()` – this returns the value of the now field but modifies the internal state so that any subsequent attempt to use `x()=t` will result in a runtime exception.

`ClockedAcc[T]` has a constructor that takes two `T` values and a `Reducer[T]` as argument. The two `T` values are used to initialize the current and next fields. These arguments should be "new" (that is, no other data-structure should have a reference to these arguments). The reducer is used to perform accumulate operations.

Operations for `x:ClockedAcc[T]`:

- `x()` – this returns the value of the now field.
- `x() <- t` – this accumulates `t` into the next field. Note: No write-write conflicts are possible.
- `x() = t` – this resets the value of the next field to `t`. To avoid read-/write and write/write conflics, this operation should be invoked only by the closure argument of `Clock.advanceAll(closure)`. (See below.)
- `x.finalized()` – this returns the value of the now field but modifies the internal state so that any attempt to use `x()=t` or `x() <- t` will result in a runtime exception.

We add the following method on Clock:

```
public static def advanceAll(x:()=>void) {...}
```

If all activities registered on the clock invoke `advanceAll(f)` (for the same value `f`), then `f` is guaranteed to be invoked by some activity A registered on the clock at a point in time when all other activities have entered the `advanceAll(f)` call and the current/next swap has been performed for all registered clocked values. At this point – also called the *clock quiescent point* – it is guaranteed that none of the other activities are performing a read or write operation on user-accessible memory.

(A possible implementation of `Clocked[T]` and `ClockedAcc[T]` is that a system-synthesized closure (that performs the current/next swap) is run at the clock quiescent point before the user specified closure is run.)

Example 3.5. `@det`

```
def stencil(a:Array[Double], eps:Double, P:Int) {
    val red = new Reducible[Double]() {
            public def zero()=0.0D;
            public operator this(x:Double, y:Double)
    = Math.max(x,y);
        };
    val err = new ClockedAcc[Double](Double.MAX_VALUE,
       Double.MAX_VALUE, red);
    val b = new Clocked[Array[Double](1)](
               new Array[Double](a.region,
    (p:Point(a.rank))=>0.0D);
               new Array[Double](a.region,
    (p:Point(a.rank))=>a(p)));

    clocked finish
       for (myRegion in a.region.partition(P))
       clocked async {
       while (err() > eps) {
         for (k in myRegion) {

            val ck = (b()(k-1)+b()(k+1))/2;

            err() <- Double.abs(ck - b()(k));

            // @Write invocation on next. Det because
    each async
            // writes into its myRegion and each
    element of the array
            // of regions produced by
    a.region.partition(P) is disjoint
            // from the other.

            b()(k) = ck;
         }
         Clock.advanceAll();
       }
    }
    return b.finalized();
}
```

In the example above there is no need to reinitialize the value of `err()` between phases since the value will monotonically decrease. However for some other computations this may be necessary. For example, suppose the error metric was the sum of all errors. The the above code would change as follows. We would pass in a different reduction operation red that sums rather than returns a max. Further, we would replace the `Clock.advanceAll()` call with

```
Clock.advanceAll(()=>{err()=0.0D;});
```

This resets the next value of err to be `0.0D` before the accumulations start to happen.

4. Implementation Considerations

Registration of accumulators with activities needs to be implemented efficiently. This may require the implementation of an activity stack, with registration information being looked up lazily and cached, rather than pushed eagerly. Also this information is clearly not needed in the body of async's that can be statically analyzed to not contain accumulator operations.

5. Semantics

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`.

Overview of formalism

5.1 Syntax

Fig. 1 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`.

$\mathtt{P} ::= \overline{\mathtt{L}}, \mathtt{S}$	Program.
$\mathtt{L} ::= \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{F}};\ \overline{\mathtt{M}}\ \}$	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.

Figure 1. 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. 2.

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.

5.2 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 $\mathit{cooked}_H(\mathtt{l})$) if its map is total, i.e., $H(\mathtt{l}) = \mathtt{C}(F) \qquad \mathrm{dom}(F) = \mathit{fields}(\mathtt{C})$.

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 $\phi \vdash +\mathtt{l}$ then `l` is cooked in H and if $\phi \vdash +\mathtt{l.f}$ then $H(\mathtt{l}) = \mathtt{C}(F)$ and $F(\mathtt{f})$ is cooked in H.

The reduction relation is described in Figure 2. An S-configuration is of the form $\mathtt{S}, H$ 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 $\mathtt{e}, H$ 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 $\mathtt{l}, H$ 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` by permitting `S` to make a step even if it is preceded by `async {S1 }`. 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 ($\mathtt{f}_i$ is contained in $\overline{\mathtt{f}}$).

5.3 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 $\mathtt{l} = H(F(f))$ is cooked ($\mathit{cooked}_H(\mathtt{l})$). 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 $\mathtt{l} \in \mathrm{dom}(H)$, we have $H(\mathtt{l}) = \mathtt{C}(F), H'(\mathtt{l}) = \mathtt{C}(F')$, 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 $\vdash \mathtt{S}$) if it does not contain any free variables. We say that `S` is *annotable* if there exists ϕ, ψ such that $\phi\ \mathtt{S}\ \psi$ can be established.[1]

We say that a program $\mathtt{P} = \overline{\mathtt{L}}\mathtt{S}$ is *proper* if it is well-typed and each method in `L` can be decorated with pre-post assertions (ϕ, ψ), and `S` is annotable. 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 $\mathtt{def\ m}(\overline{\mathtt{x}} : \overline{\mathtt{C}})\{\mathtt{S}\}$ with assertion (ϕ, ψ) that it is the case that the free variables of ϕ, ψ are contained in $\mathtt{this}, \overline{\mathtt{x}}$, and that $\phi\ \mathtt{S}\ \psi$.

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

Theorem 5.1. ***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 5.2. ***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.

6. Related Work

DPJ

Phasers

CCP

7. Conclusion

We show that many determinate concurrent programs can be written in X10 using determinate, deadlock-free constructs, so that they are determinate by design.

Acknowledgments

This material is based in part on work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] An example of a statement that is *not* annotable 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.

$$\frac{\texttt{S}, H \leadsto H'}{\begin{array}{l}\texttt{finish \{S\}}, H \leadsto H' \\ \texttt{async \{S\}}, H \leadsto H' \\ \texttt{S S}', H \leadsto \texttt{S}', H'\end{array}}\ \text{(R-Term)} \qquad \frac{\texttt{S}, H \leadsto \texttt{S}', H'}{\begin{array}{l}\texttt{finish \{S\}}, H \leadsto \texttt{finish \{S}'\texttt{\}}, H' \\ \texttt{async \{S\}}, H \leadsto \texttt{async \{S}'\texttt{\}}, H' \\ \texttt{S S}_1, H \leadsto \texttt{S}'\ \texttt{S}_1, H' \\ \texttt{async \{S}_1\texttt{\} S}, H \leadsto \texttt{async \{S}_1\texttt{\} S}', H'\end{array}}\ \text{(R-Step)} \qquad \frac{\texttt{e}, H \leadsto \texttt{l}, H'}{\texttt{val x = e;S}, H \leadsto \texttt{S}[\texttt{l}/\texttt{x}], H'}\ \text{(R-Val)}$$

$$\frac{\texttt{l}' \notin \mathrm{dom}(H)}{\texttt{new C}, H \leadsto \texttt{l}', H[\texttt{l}' \mapsto \texttt{C}()]}\ \text{(R-New)} \qquad \frac{H(\texttt{l}') = \texttt{C}(\ldots) \qquad \mathit{mbody}(\texttt{m},\texttt{C}) = \overline{\texttt{x}}.\texttt{S}}{\texttt{l}'.\texttt{m}(\overline{\texttt{l}}), H \leadsto \texttt{S}[\overline{\texttt{l}}/\overline{\texttt{x}}, \texttt{l}'/\texttt{this}], H}\ \text{(R-Invoke)}$$

$$\frac{H(\texttt{l}) = \texttt{C}(\overline{\texttt{f}} \mapsto \overline{\texttt{l}'})}{\texttt{l.f}_i, H \leadsto \texttt{l}'_i, H}\ \text{(R-Access)} \qquad \frac{H(\texttt{l}) = \texttt{C}(F) \qquad \mathit{cooked}_H(\texttt{l}')}{\texttt{l.f} = \texttt{l}', H \leadsto H[\texttt{l} \mapsto \texttt{C}(F[\texttt{f} \mapsto \texttt{l}'])]}\ \text{(R-Assign)}$$

Figure 2. FX10 Reduction Rules ($\texttt{S}, H \leadsto \texttt{S}', H' \mid H'$ and $\texttt{e}, H \leadsto \texttt{l}, H'$).