Lightweight Safe Concurrent Programming

Abstract

The safe parallelization of sequential programs is an important and challenging task. A parallelization is safe if it ensures that no new behaviors are introduced; the semantics of the parallel program is identical to that of the sequential program. In particular, the parallel program must not exhibit scheduler indeterminacy or deadlock.

This is a challenging problem. Imperative memory locations are famously indeterminate under concurrent read/write operations. Further, most concurrent programming languages support synchronization mechanisms that enable a thread to wait until some appropriate condition is true, thus leading to the possibility of deadlock.

In this paper we develop a very rich fragment of the concurrent, imperative programming language X10 in which *all* programs have sequential semantics. The advantages of such a language are striking: the programmer simply cannot introduce indeterminacy or deadlock-related errors. All programs can be debugged as if they are sequential programs, without having to reason about all possible interleaving of threads.

The fragment includes all of X10's most powerful constructs – (clocked) `finish` and `async`, places and `at`. We introduce two new abstractions – *accumulators* and *clocked values* – with lightweight compiler and run-time support. Accumulators of type `T` permit multiple concurrent writes, these are reduced into a single value by a user-specified reduction operator. Accumulators are designed to be determinate and deadlock-free at run-time, under all possible uses. Clocked values of type `T` operate on two values of `T` (the *current* and the *next*). Read operations are performed on the current value, 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" idiom. Clocked types are designed in such a way that they are safe even in the presence of arbitrary aliasing and patterns of concurrency. Non-accumulator clocked types require the compiler to implement a lightweight "owner computes" rule to establish disjointness of writes within a phase.

We show all programs written in this fragment of the language are guaranteed to have sequential semantics. That is, programs in this fragment may be developed and debugged as if they are sequential, without running into the state explosion problem.

Further, many common patterns of concurrent execution (particularly array-based computations) can be written in this language in an "obvious" way. These patterns include histograms, all-to-all reductions, stencil computations, master/slave execution. We show that there are simple statically-checkable rules that can establish for many common idioms that some run-time synchronization checks can be avoided.

The key technical problem addressed is unrestricted aliasing in the heap. We develop the idea of an *implicit ownership domain* (first introduced in the semantics of clocks in X10 in [31]) that newly created objects may be registered with the current activity so that they can be operated upon only by this activity and its newly spawned children activity subtree. This permits safety to be established locally, independent of the context in which the code is being run.

1. Introduction

A *serial* schedule for a parallel program is one which always executes the first enabled step in program order. A *safe* parallel program is one that can be executed with a serial schedule *S* and for which for every input every schedule produces the same result (error, correct termination, divergence) as *S*. Such a program is semantically a sequential program, hence it is scheduler-determinate and deadlock-free.

A *safe parallel programming language* is an imperative parallel programming language in which every legal program is a safe program. Programmers can write code in such a language secure in the knowledge that they will not encounter a large class of parallel programming problems. Such a language is particularly useful for parallelizing sequential (imperative) programs. In such cases (*contra* reactive programming) the desired application semantics are sequential, and parallelism is needed solely for efficient implementation.

The characteristic property of a safe programming language is that the *same* program has a sequential reading and a parallel reading, and both are compatible with each other. Hence the program can be developed and debugged as a sequential program, using the serial scheduler, and then run the unchanged program in parallel. Parallel execution is guaranteed to effect only performance, not correctness. Safety is a very strong property.

One way to get safety is through implicitly parallel languages (e.g. Jade [28],IPOT [36]). One starts with a sequential programming language, and adds constructs (e.g. tasks) that permit speculative execution while guaranteeing that the only observable write to a shared variable is the write by the last task to execute in program order. While this work is promising, extracting usable parallelism from a wide variety of sequential programs remains very hard.

Explicitly parallel programming languages provide a variety of constructs for spawning tasks in parallel and coordinating between them. Here the programmer can typically directly control the granularity of concurrency, and locality of access (e.g. placement of data-structures in a multi-node computation) and use efficient concurrent primitives (atomic reads/writes, test and sets, locks etc) to control their execution.

For such languages proving that a program is safe – much less that the programming language is safe – now becomes very hard, particularly for modern object-oriented languages which allow the programmer to create arbitrarily complicated data-structures in the shared heap. It becomes very difficult to show that any possible schedule will produce the same result as the serial schedule.

Our starting point is the language X10 [30] since it offers a simple and elegant treatment of concurrency and distribution, with some nice properties. In brief, X10 introduces the constructs `async S` to spawn an activity to execute `S`; `finish S` to execute `S` and wait until such time as all activities spawned during its execution have terminated; `at (p) S` to execute `S` at place `p`. These constructs can be nested arbitrarily – this is a source of significant elegance and power. Additionally, X10 v 2.2 introduces a simplified version of X10 clocks (adequate for many practical usages) – `clocked finish S` and `clocked async S`. Briefly, a clocked finish introduces a new barrier that can be used by this activity and its children activities for synchronization.[1]

[31] establishes that a large class of programs in X10, namely those that use `finish`, `async` and `clock`s are deadlock-free. The central intuition is that a clock can only be used by the activity that created it and by its children, and hence the spawn tree structure can be used to avoid depends-for cycles.

To understand the conditions under which such programs may be safe we need to understand what it means to execute such a program with a serial schedule. In a serial schedule, the statement `async {S1} S2` is executed by first executing `S1`, and then, on its completion, executing `S2` (since the first enabled step of `async {S1} S2` in program order will be the first enabled step of `async {S1}`. That is, `async S1` behaves like `S1`. Similarly, `finish {S1} S2` is executed by first executing `S1` to completion and then executing `S2`. Thus under a serial schedule, `finish` and `async` behave like "no-ops".

The situation is slightly different for `clocked finish` and `clocked async` which can have the special `advance` statement to go to the next phase of the clock.[2] The first enabled step of `clocked finish {S1} S2` is going to be the first step of `S1`. However, the first enabled step of `clocked async {S1} S2` may be the first step of `S2` – e.g. when `S1` is `advance; S3` and the clocked async must wait until all other such asyncs in the scope of the clocked finish are ready to advance. Thus a `clocked finish {S}` exhibits "round-robin" scheduling of `S`. For instance, through this reasoning we can establish that the serial schedule for

```
clocked finish {
  clocked async { S1; advance; S3}
  clocked async { S2; advance; S4}
}
S5
```

would execute in the order: `S1;S2;S3;S4;S5`.

When is such a program safe? The central problem is to ensure that in the statement `[clocked] async {S} S1` it is not the case that `S` can write to a location that `S1` can read from or read from a location that `S` can read from. Otherwise the behavior is not equivalent to a serial schedule. One line of attack has been the pursuit of *effect systems* [23], [22], [9], [8]. Broadly, static effects systems call for a user-specified partitioning of heap into *regions* in a fine-grained enough way to show that operations that may occur simultaneously work on different regions. For instance [8] introduces separate syntax for regions, introduces the ability to specify an arbitrary tree of regions, and new syntax for specifying which region mutable locations belong to. Methods must be associated with their read and write effects which capture the set of regions that can be operated upon during the execution of this method. Now determinacy can be established if it can be determined that in `async {S} S1` it is the case that `S` does not write any region that `S1` reads or writes from, and vice versa (*disjoint parallelism*).

[8] develops these ideas in the context of a language with `cobegin/coend` and `forall` parallelism, and does not address arbitrary nested or clocked parallelism. In particular it is not clear how to adapt these ideas to support some form of pipelined communication between multiple parallel activities. Once can think of these computations as requiring "disjointness across time" rather than disjointness across space. A producer is going to write into locations `w(t),w(t+1),w(t+2), ...`, but this does not conflict with a consumer reading from the same locations as long as the consumer can arrange to read the values in time-staggered order, i.e. read `w(t)` once the produce is writing `w(t+1)`.

We believe it is possible to significantly simplify the approach of [8] (e.g. using the dependent-type system of X10, see Sec. 6) and extend it to cover more X10 concurrency constructs. Nevertheless the cost of developing these region assertions is not trivial. Their viability for developing large commercial-strength software systems is yet to be established.

1.1 Lightweight safety

In this paper we chose a different approach, a *lightweight* approach to safety. We introduce two simple ideas – *accumulators* and *clocked types* – which require very modest compiler support and can be implemented efficiently at run-time. We show that programs written using accumulators and clocked types as the only shared variables are safe.

Accumulators arise very naturally in concurrent programming: an accumulator is a mutable location associated with a commutative and associative *reduction* operator that can be operated on simultaneously by multiple activities. Multiple values offered by multiple activities are combined by the reduction operator. We show that the concrete rules for accumulators can be defined in such a way that they do not compromise safety: a serial execution precisely captures results generable from any execution. This is possible even though an accumulator is permitted to go through many phases of accumulations, with its value being observable between these phases.

Similarly clocked computations (barrier-based synchronization) are quite common in parallel programming e.g. in the SPMD model, BSP model etc. We observe that many clocked programs can be written in such a way that shared variables take on a single value in one phase of the clock. Further, clocked computations are often iterative and operate on large aggregate data-structures (e.g. arrays, hash-maps) in a data-parallel fashion, reading one version of the data-structure (the "red" version) while simultaneously writing another version (the "black" version). To support this widely used idiom, we introduce the notion of *clocked types*. An instance `a` of a clocked type `Clocked[T]` keeps two instances of type `T`, the `current` and the `next` instance. `a` can be operated upon only by activities registered on the current clock. All read operations during the current clock phase are directed to the `current` version, and write operation to the `next` version. (Read operations on `T` must be marked as `@Read`, and write operations as `@Async` or as `@Write` based on whether they are accumulative.) This ensures that there are no read-write conflicts. Invocations of `@Write` methods are subject to an "owner computes" rule (lightweight static effects checking): in a parallel loop `for i in R) async S`, `S` may update only the `i`th element of an array. Once computation in the current phase has quiesced – and before activities start in the next phase – the `now` and `next` versions are switched; `now` becomes `next` and `next` becomes `now`.[3]

We show that clocked types are safe. The only dynamic check needed is that values of this type (and the values of type `T` passed

[1] X10 also has a conditional atomic construct, `when (c) S` which permits data-dependent synchronization and can introduce deadlocks. We do not consider this construct in this paper.

[2] In the paper we use `advance` as a keyword, however, in X10 one need to use the `Clock.advance()` method.

[3] Clearly this idea can be extended to k-buffered clocked types where each clock tick rotates the buffer. This idea is related to the K-bounded Kahn networks of [14] and multi-version variables of John Mellor-Crummey.

into it on creation) are being operated upon only by the activity that created the value or its descendants.

In the following by Safe X10 we shall mean the language X10 restricted to use (`clocked`) `finish`/`async`, `at` with accumulators, or with clocked types, and with the "owner computes" rule checked statically for write-access to elements of clocked types. All programs in Safe X10 are safe – they can be run with a serial schedule and their I/O behavior is identical under any schedule. We show that Safe X10 is surprisingly powerful. Many concurrent idioms can be expressed in this language – histograms, all-reduce, and master/slave communication. Indeed, even some form of pipelined/systolic communication is expressible. This is a novel feature of our proposal – past work such as [33] cannot deal with such clocked computation.

Since the dynamic conditions introduced on accumulators and clocked types are not straightforward, we formalize the concurrent and serial semantics of an abstraction of Safe X10 using Plotkin's structural operational style. We are able to do this in such a way that the two proof systems share most of the proof rules, simplifying the proof. We establish that the language is safe – for any program and any input, any execution sequence for the concurrent proof rules can be transformed into an execution sequence for the sequential proof rules with the same result.

In summary the contributions of this paper are as follows.

- We identify the notion of a *safe* program – one which can be executed with a serial schedule and for which every schedule produces the same result. Such a program is simultaneously a sequential program and a parallel program with identical I/O behavior.
- We introduce accumulators and clocked types in the X10 programming model. These are introduced in such a way that arbitrary programs using (`clocked`) `finish`, `async` and `at` and in which the only variables shared between concurrently executing activities are accumulators or clocked types are guaranteed to be safe.
- We show that many common programming idioms can be expressed in this language.
- We formalize a fairly rich subset of X10 – including (clocked) finish, async, accumulators and clocked accumulators. This is the first formalization of the nested clock design of X10 2.2, and is substantially simpler than [31]. We establish that this language is safe.

In companion work we show how these ideas can be extended to support modularly defined effects analysis, using X10's dependent type system.

The rest of this paper is as follows. In Sec. 2 we present the constructs in detail, followed by examples of their use in Sec. 3. Sec. 4 presents the semantics of these constructs. In Sec. 5 we discuss related work. We discuss implementation in Sec. 6 and finally conclude with future work.

2. Accumulators and Clocked Types

2.1 X10

The X10 construct `async S` spawns a new activity to execute `S`. This activity can access variables in the lexical environment. The construct `finish S` executes `S` and waits for all asyncs spawned within `S` to terminate.

X10 2.2 introduces *clocked* versions of `async` and `finish`. This design (which is the one we will consider in this paper) is a simplification of the original design and has the nice property that clocks are not present as data objects in the source program, thus removing a potential source of error.

Clocks are motivated by the desire to support barrier computations in which all threads in a given group must reach a point in the code (a barrier) before all of them can progress past it. X10 permits many clocks to exist at the same time, permits an activity to operate on multiple clocks at the same time, and permits newly created activities to operate on existing clocks.

X10 2.2 permits `async S` and `finish S` to be optionally modified with a `clocked` qualifier. When an activity executes `clocked finish S` it creates a new clock `c`, registers itself on it, and executes `finish S` with `c` as the *current clock*. The clock object is implicit in that it cannot be referred to in the code. Within `S`, the `clocked async S` construct can be used to spawn an async registered on the current clock; `async S` should be used to spawn an async that is *not* registered on the current clock.

Note that `finish` blocks can be arbitrarily nested, so an activity may at any given time be registered on a stack of clocks. The current clock is always the one most recently pushed on to the stack. No support is provided to register an activity on an ancestor clock.

When a clocked async terminates it automatically de-registers from the clock; when the parent activity that entered the `clocked finish` block reaches the end of the block it also automatically de-registers from the clock.

The special statement `advance` can be used to advance the current clock. Whenever an activity hits `advance` it blocks on the current clock it is registered with. (It is illegal for an un-clocked `async` to execute `advance`.) Once all activities registered on the clock have reached an advance, all of them can progress.

Additionally, X10 permits computations to be run on multiple places. A place consists of data as well as activities that operate on it. Typically a place is realized as an operating system process in a cluster or multi-node computer. The `at (p) S` construct is used to cause the current activity to shift to place `p` and execute `S`.

`at` blocks and (`clocked`) `async`, `finish` blocks can be arbitrarily nested and arbitrarily combined with recursion. That is, method invocations can return having spawned `async`s that have not terminated. The body `S` of a `clocked finish S` may spawn an `async` that is not clocked, or may spawn a nested `clocked finish S`. This flexibility makes X10 a very succinct and elegant language for expressing many patterns of communication and concurrency.

2.2 Accumulators

The class `Acc[T]` implements the notion of an accumulator. To construct an instance of this class a binary associative and commutative operator `f` over `T` (the *reduction operator*) must be supplied, together with the zero for the operator, `z` (satisfying `f(v,z)=z` for all `v`), and an initial value from `T`. The accumulator provides the following operations. For `a` such a value, the operator `a() <- v` accumulates the value `v` into `a`. The operator `a()` returns the current value. The operator `a() = v` sets the value of the accumulator to `v`.

There are no restrictions on storing accumulators into heap data-structures, or aliasing them. However, each accumulator is *registered* with the activity that created it. This registration is automatically inherited by all activities spawned (transitively) by this activity. It is an error for an activity to operate on an accumulator it is not registered with. Thus one can 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. In particular, accumulators can be passed into arbitrary method invocations, returned from methods etc, transmitted to other places with no restrictions. The only restriction is that the accumulator can only be used by the activity that created it or its children.

There are no restrictions on the use of the accumulate operation `a()<- v`. The same activity can perform multiple accumulate operations on one or more accumulators. Thus, the same set of spawned activities can perform multiple Map Reduce patterns simultaneously.

However, the read operation `a()` and the update operation `a()=v` are subject to a dynamic restriction. (1) They may only be performed

by the activity that created a. (2) Say that an activity A is in *synchronous mode* if all asyncs spawned by A have terminated or stopped at a next on a clock registered on A. In such a situation there is no concurrent mutator of a (an activity not in the subtree of A that may have acquired a reference to a through the heap is unable to perform any operation on it). Hence it is safe to read a. Similarly for update. Therefore the semantics of these operations require that the operation *suspend* until such time as the synchronous condition is true.

Intuitively, this does not introduce any deadlocks since the spawned activities cannot be waiting for any operation by the parent (other than an advance) in order to progress. It clearly preserves serial semantics since a read will only reflect the results of accumulations performed earlier in program order. A more detailed treatment is provided in Sec. 4.

In many cases the compiler can statically evaluate whether an activity is registered on the clock and/or whether a synchronous call will suspend. In such cases the checks can be eliminated.

Example 2.1. *In the following example, 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 in* x()*.*

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

2.3 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 objects are registered with activities, just like accumulators. This ensures that any other activity with access to the clocked object (or its encapsulated objects) throught he heap is unable to operate on them. Only the activity that created a Clocked[T] value v can enter a clocked finish S block within which v can be operated on by multiple activities. Such a block is said to be the *current block* for v. Only and precisely the clocked asyncs spawned dynamically at the top-level in the body of the current block can operate on v. In particular if a top-level clocked async were to further spawn a clocked finish S1, then v would not be considered accessible within S1 (since the current block and current clock have changed). This is checked dynamically.

Similarly, v is not accessible within a async S1 spawned dynamically by S since S1 operates asynchronously to the clock and hence cannot access v determinately. This is also checked dynamically. Note that a clocked finish S can advance on its clock even if S dynamically spawns an un-clocked async that is active. The only requirement to advance the clock is that all clocked asyncs registered on the clock have reached an advance. These rules imply that the activity that created v can operate on v in multiple successive clocked finish statements, with arbitrary sequential code (accessing v) in between.

Clocked[T] has a constructor that takes two T arguments, these 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), and are considered registered with the creating async and its descendants.

For x:Clocked[T] the following operations available:

- x(): this returns the value of the current field. Note that this operation can be called simultaneously by multiple activities (unlike Acc).
- x() = t: This sets the value of the next field to t. Note: write-write conflicts are possible since multiple activities may try to set the value at the same time. So a lightweight static technique should be used to establish that at most one activity is writing to that location. In the case of arrays of clocked types this can often be established using an "owner computes" rule.

Further the compiler treats invocations of methods defined on T and invoked on x() specially. A method may be marked @Read – such a method must not mutate any state (this is checkedby the compiler). Invocations of such methods are performed on the current field. For instance, for arrays, the accessor method is marked @Read. A method may also be marked @Write or @Async – these methods may mutate state and are invoked on the next field. The first kind of method must be called by at most one activity in a phase (and is subject at compile-time to the owner computes rule), the second may be called by more than one activity. The first is permitted to update state, the second may not, but may invoke reduction operations. For arrays and accumulators, the updater method is marked @Write. For accumulators, the accumulate method is marked @Async.

We add the following method on Clock:

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

If all activities registered on the clock invoke advance(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 advance(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. [4]

3. Programming Examples

We now demonstrate that several common idioms of concurrency and communication lie in Safe X10. These programs are X10 programs where methods and statements intended to be safe are marked with a @safe annotation, thereby triggering appropriate compiler checks.

3.1 Example use of Acc

Example 3.1 (Histogram)**.** *The histogram problem is easily represented with a* Rail *of accumulators:*

```
@safe
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));
}
```

Note that programs that accumulate a figure of merit can be written with accumulators. For instance the N-Queen program needs to use a single accumulator for collecting the result. Indeed it can be written with the even more restrictive collecting finish style

[4] A possible implementation of Clocked[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.

discussed below. Similarly for Monte-Carlo simulations in which the main activity spawns several children activities to perform a simulation, and the simulation returns results which can be accumulated to obtain the final answer. We have also written a mutiplace implementation of Bader and Ja Ja median finding algorithm in Safe X10, using accumulators and clocked types.

Map Reduce applications such as word-count (that do not require intermediate sorting of results) can also be expressed directly:

Example 3.2 (Distributed word-count)**.** *Here the difference from histogram is that the accumulation may be done at a remote place.*

```
@safe
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 distributed reductions in a straightforward "shared memory" style.

Example 3.3 (Reduction)**.** *Here we show the single-sided, blocking version. Spawned activities update an accumulator created by the parent activity, which reads it after a* `finish`*.*

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

***Collecting* `finish`** The collecting `finish` construct is of the form `finish(r) S`, where `r` is an instance of `Reducible[T]` and `S` is a statement. Within the dynamic execution of `S`, any execution of the statement `offer t;` results in a value `t` being accumulated in a set. (`t` must be of type `T`.) The result of the reduction of this set of `T` values with `r` is then returned.

```
{
  val x = new Acc[T](r);
  finish {
    S [ x <- t / offer t]
  }
  x()
}
```

An attractive aspect of collecting finish is that nesting is used quite naturally to reflect the relationship between the parallel computation performing the accumulation and the value returned. In particular there is no need to explicitly introduce the notion of an accumulator, or to register it with the current block, or to check that other activities in the current block have quiesced.

On the other hand, this strength is also a limitation. It is not possible to use the same idiom for clocked code, i.e. collect values being offered by multiple `clocked` asyncs in the current phase. Further, using this idiom safely across method calls requires the addition of `offers T` clauses (similar to `throws T`) clauses, specifying the type of the value being offered. Otherwise at run-time a value of an incompatible type may be offered leading to a run-time exception. Finally, the lack of a name for the result being collecting means that it is difficult to use the same computation to accumulate multiple separate values without more contortions. E.g. one could set the return type to be a tuple of values, but then the `offer` statement would need to specify the index for which the `offer` was intended. Now it is not clear that the index type should be arithmetic – why should not one be able to collect into the range of a `HashMap` (so the index type could be an arbitrary type `Key`)?

We feel that these decisions are all orthogonal to the actual process of accumulating and should be dealt with by the data-structuring aspects of the language design.

3.2 Clocked computations

Example 3.4 (Pipeline)**.** *This example creates* 3 *stages in parallel. Stage* 1 *gets the input (in this case generating the natural numbers), Stage* 2 *increments it by* 1*, Stage* 3 *doubles it and prints the value. An immutable array of clocked values forms the pipeline.*

```
@safe
def pipeline() {
    val a = [new Clocked[Int](0,0),
             new Clocked[Int](0,0),
             new Clocked[Int](0,0)];
    clocked finish {
       for (i in a) {
        if (i==0)
          clocked async {
            for (var j:Int=0;; i++) {
                a(i)()=j;
                advance;}}
        if (i==1)
          clocked async {
            advance;
            for (;;) {
                a(i)()= a(i-1)()+1; //1,2,3,...
                advance;}}
        if (i==2)
          clocked async {
           advance;
           advance;
           for(;;) {
              a(i)()= 2*a(i-1)();
              advance;
              Console.OUT.println(a(i)()); //2,4...
        }}}}}
```

Note the serial schedule generates:

```
a(0)()=0; advance;
a(0)()=1; a(1)()=1; advance;
a(0)()=2; a(1)()=2; a(2)()=2; advance; //print 2
a(0)()=3; a(1)()=3; a(2)()=4; advance; //print 4
...
```

This shows the pipelined nature of the computation.

Example 3.5 (Butterfly idiom)**.** *This program uses a butterfly idiom to compute an all to all reduction.*

```
def reduce[T](da:DistArray[T], red:Reducible[T]) {
  val P = Place.MAX_PLACES;
  val phases = Utils.log2(P);
  val x = new DistArray[Clocked[Acc[T]]](da.dist,
     (p:Point(da.rank))=>new Clocked[Acc[T]](new
     Acc[T](da(p),red),
        new Acc[T](da(p),red)));
  clocked finish {
    for (p in x.dist.places()) clocked async at(p) {
      var shift:Int=1;
      for (phase in 0..phases-1) {
```

```
        x((p.id+shift)%P)() <- x(p.id)();
        shift *=2;
        advance;}}}
  return x(0)();
}
```

Example 3.6. *A stencil computation can be performed with a* `Clocked[Acc[T]]` *to accumulate errors, and with a* `Clocked[Array[Double]]` *to store the array being operated on. Note that this program requires a a clocked array of doubles rather than a clocked array of accumulators for doubles because at each iteration the newly computed value must overwrite the value in the array element, not accumulate into it.*

Hence to establish safety some additional information is needed, such as `partition` *partitions* `b.region` *into* `P` *pieces and hence the writes are all disjoint.*

```
def make(r:Region)=new Array[Double](r,
     (p:Point(r))=> 0.0D);
def stencil(a:Array[Double], eps:Double, P:Int) {
 val red = Double.Reducible.max;
 val err = new Clocked[Acc[Double]](new
     Acc[Double](0.0,red),
    new Acc[Double](0.0, red));
 val b = new Clocked[Array[Double](1)](make(a.region),
    make(a.region));
 clocked finish
  for (myRegion in b.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));
     b()(k) = ck;
    }
    Clock.advance( ()=> { err()()=0.0; });}}}
 return new Array[Double](a.region,
    (p:Point(a.region))=>b(p));}
```

4. 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`, `async`, clocks and accumulators.

Overview of formalism We present the reduction rules for two schedulers: a concurrent and a sequential scheduler. The sequential scheduler is obtained from the concurrent one by removing two rules. Intuitively, the sequential scheduler performs a DFS over `async`. We prove that the concurrent and sequential schedulers always have the same result (i.e., isomorphic heaps). Sec. 4.1 presents FX10 syntax, Sec. 4.2 the reduction rules, and Sec. 4.3 our main result that programs in FX10 are always safe.

4.1 Syntax

There are three predefined classes in FX10: `Object`, `Acc` (representing accumulators), and `ClockedAcc` (taken as shorthand for `Clocked[Acc[T]]`). X10 has an effect system that guarantees that `async` only accesses immutable state, clocked values and accumulators. The effect system also ensures that clocked values can only be assigned once every phase, however clocked accumulators can be assigned multiple times. FX10 only models the runtime behavior without the effect system. Therefore, in order to guarantee determinacy without an effect system, FX10 does not model field assignment nor general clocked values (only clocked accumulators). Objects are still mutable because accumulators and clocked accumulators are mutable.

$\texttt{P} ::= \overline{\texttt{L}}, \texttt{S}$	Program.
$\texttt{L} ::= \texttt{class C extends D \{ } \overline{\texttt{F}}\texttt{; } \overline{\texttt{M}} \texttt{ \}}$	cLass declaration.
$\texttt{F} ::= \texttt{var f : C}$	Field declaration.
$\texttt{M} ::= \texttt{def m}(\overline{\texttt{x}} : \overline{\texttt{C}}) : \texttt{C}\{\texttt{S}\}$	Method declaration.
$\texttt{p} ::= \texttt{l} \mid \texttt{x}$	Path.
$\texttt{e} ::= \texttt{p.f} \mid \texttt{new C}(\overline{\texttt{p}}) \mid \texttt{p}()$	Expressions.
$\texttt{B} ::= (\overline{\texttt{l}})\{\texttt{S}\}$	Blocks.
$\texttt{S} ::= \varepsilon \mid \texttt{advance;} \mid \texttt{p.m}(\overline{\texttt{p}})\texttt{;} \mid$ $\texttt{val x = e;} \mid \texttt{p} \leftarrow \texttt{p} \mid \texttt{S S} \mid$ $\texttt{[clocked] [finish \mid async] B}$	Statements.

FX10 Syntax. The terminals are locations (`l`,`a`), parameters and `this` (`x`), field name (`f`), method name (`m`), class name (`B`,`C`,`D`,`Object`), and keywords. The program source code cannot contain locations (`l`), because locations are only created during execution/reduction in R-NEW of Fig. 1.

Fig. 4.1 shows the abstract syntax of FX10. A constructor in FX10 assigns its arguments to the object's fields.

ε is the empty statement. Sequencing associates to the left. The block syntax in source code is just $\{\texttt{S}\}$, however at run-time the block is augmented with (the initially empty) set of locations π representing the objects registered on the block, which are those that were created in that block.

Statement $\texttt{val x = e;S}$ evaluates `e`, assigns it to a new variable `x`, and then evaluates `S`. The scope of `x` is `S`.

Statement $\texttt{p}_1 \leftarrow \texttt{p}_2$ is legal only when $\texttt{p}_1$ is an accumulator, and it accumulates $\texttt{p}_2$ into $\texttt{p}_1$ (given the reduction operator that was supplied when the accumulator was created). Expression $\texttt{p}()$ is legal only when `p` is an accumulator; when the accumulator is clocked it immediately returns the old value of the accumulator, and when it is not clocked it waits until all activities created in the block have terminated or are ready to advance (then it is a safe point to read the accumulator because it cannot be further mutated).

4.2 Reduction

A *heap* H is a mapping from a given set of locations to *objects*. An object is a pair $\texttt{C}(\overline{\texttt{l}})$ where `C` is a class (the exact class of the object), and $\overline{\texttt{l}}$ are the values in the fields.

An *configuration* is of the form $\texttt{S},\texttt{H}$ 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 a successfully terminated computation). The reduction rules define transitions between configurations. We define both a concurrent scheduler, and by removing two rules we also define a sequential scheduler.

The reduction relation $\rightsquigarrow_\pi$ is described in Fig. 1. Here π is a set of locations which can currently be asynchronously accessed. Thus each transition is performed in a context that knows about the current set of capabilities.

Fig. 1 also defines the *advance relation* $\texttt{S} \overset{c,a}{\dashrightarrow} \texttt{S}'$ that is used to advance a statement to its next phase. The integer c represent how many `clocked async` can advance to the next phase in `S`, and the integer a represent how many non-nested `async` are in `S`. For example, the statement

```
async { S1 }
clocked async {
  clocked async {}
  async { S2 }
  clocked async { advance; S3 }
  advance; S4
}
clocked async { advance; S5 }
```

can be advanced to (where $c = 3$ and $a = 2$)

```
async { S1 }
```

```
clocked async {
  clocked async {}
  async { S2 }
  clocked async { S3 }
  S4
}
clocked async { S5 }
```

We write $\texttt{S} \xrightarrow{c,a} \_$ if the result of advancing S is not used in the rule (i.e., _ is the statement after advancing S, but it is unused.) Finally, we write $\texttt{S} \xrightarrow{c+,a+} \texttt{S}'$ if $\texttt{S} \xrightarrow{c',a'} \texttt{S}'$ and $c' \geq c$ and $a' \geq a$. For example, (R-SEQ)+ uses $\texttt{S} \xrightarrow{1+,0} \_$, meaning that S has at least one `clocked async` waiting on `advance`; and it does not have any non-clocked `async`. (This rule is used for the sequential scheduler, to allow it to progress across stuck activities. The concurrent scheduler can just use (R-ASYNC) instead.) The advance relation is also used in the following places: (i) $\texttt{S}_1 \xrightarrow{0+,0+} \_$ is used in (R-TRANS-B), (R-ADV), (R-NEW) to give context (it ensures that we can execute the statement after $\texttt{S}_1$). (ii) $\texttt{S}_1 \xrightarrow{0+,0} \_$ in (R-ACC-R) ensures that $\texttt{S}_1$ cannot progress and therefore it is a safe point to read the accumulator (because no other activity can mutate the accumulator). (iii) $\xrightarrow{1+,0+}$ in (R-ADV) guarantees that we can advance to the next phase (there must be at least one `advance`; that is removed).

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.

We now explain the rules in detail. The rules for (R-EPSILON), (R-TRANS), (R-ACCESS), and (R-INVOKE) are standard.

One minor novelty is in how `async` is defined. The critical rule is the last rule in (R-ASYNC)- – it specifies the "asynchronous" nature of `async` by permitting S to make a step even if it is preceded by `async` $\texttt{S}_1$. For the sequential scheduler, we define (R-SEQ)+ that permits S to make a step only if it is preceded by `async` $\texttt{S}_1$ that cannot progress (it has at least one `clocked async` waiting on `advance`; and no un-clocked `async`).

Further, each block $(\overline{\texttt{l}})\{\texttt{S}\}$ records the set of newly created objects $\overline{\texttt{l}}$, which we say have registered on the block. When descending into the body in (R-TRANS), the newly registered objects are added to those obtained from the environment. When descending into an `async` or `finish` in (R-TRANS-B), we need to take special care of the clocked accumulators (because they cannot be used in a non-clocked construct). Given $\overline{\texttt{l}}$ and a heap H, we define $\texttt{Acc}(\overline{\texttt{l}})$ to return only the non-clocked accumulators in $\overline{\texttt{l}}$ (dropping all the other objects). According to (R-TRANS-B), when descending into a non-clocked `async` or `finish` , we keep only the non-clocked accumulators in π (i.e., the clocked accumulators cannot be accessed). When descending into a clocked `async` we keep π unchanged. Finally, the most complicated case is descending into a clocked `finish` : then we keep all the non-clocked accumulators and also the newly created clocked-accumulators in the enclosing block (these clocked accumulators will be advanced in rule (R-ADV) later).

The rule (R-NEW) returns a new location that is bound to a new object that is an instance of C with its fields set to the argument; the new object is registered with the block. (It is enough to register only clocked-values and accumulators, but for simplicity we register all newly created objects.)

The rule (R-ACC-W) updates the current contents of the accumulator provided that the current set of capabilities permit asynchronous access to the accumulator. Rule (R-CLOCKED-W) is similar.

The rule (R-ACC-R) permits the accumulator to be read in a block provided that the current set of capabilities permit it (if not, an error is thrown), and provided that the only statements prior to the read are re is no nested async prior to the read are clocked asyncs that are stuck at an `advance`. Rule (R-CLOCKED-R) immediately returns the old value of the accumulator (without waiting like in (R-ACC-R)).

The rule (R-ADV) permits a `clocked finish` to advance only if all the top-level `clocked asyncs` in the scope of the `clocked finish` are stuck at an `advance`. Note that an un-clocked `async` may exist in the scope of the `clocked finish`; they do not come in the way of the `clocked finish` advancing. The resulting heap $\texttt{H}' = \texttt{switch}(\texttt{H}, \overline{\texttt{l}})$ is obtained from H by switching the current and next values in all clocked accumulators in $\overline{\texttt{l}}$.

4.3 Results

Our main result is that all programs in FX10 are safe, i.e., all schedulers are deadlock-free and result in the same heap (up to isomorphism of locations).

Theorem 4.1. *Given a well-typed program* $\overline{\texttt{L}}, \texttt{S}$ *in heap H, if* $\texttt{S}, H \overset{*}{\rightsquigarrow} H'$ *and* $\texttt{S}, H \overset{*}{\rightsquigarrow} H''$ *then* H' *and* H'' *are isomorphic.*

5. Related Work

5.1 Programming Models

Determinism in parallel programming is a very active area of research. Guava [4] introduces restrictions on shared memory Java programs that ensure no data-races primarily by distinguishing monitors (all access is synchronized) from values (immutable) and objects (private to a thread). However Guava is not safe since Guava programs may use `wait`/`notify` for arbitrary concurrent signalling and hence may not executable with a sequential schedule. The Revisions programming model [10] guarantees determinism by isolating asynchronous tasks but merging their writes determinately. However, the model explicitly does not require that a sequential schedule be valid (c.f. Figure 1 in [10]).

DPJ develops the "determinacy-by-default" slogan using a static type-and-effects system to establish commutativity of concurrent actions. The deterministic fragment of DPJ is safe according to the definition above. Safe X10 offers a much richer concurrency model which guarantees the safety of common idioms such as accumulators and cyclic tasks (clocks) without relying on effects annotations. The lightweight effects mechanism in X10 can be extended to support a much richer effects framework (along the lines of DPJ) using X10's constrained type system. We leave this as future work.

The SafeJava language [27] is unfortunately not safe according to our definition, even though it guarantees determinacy and deadlock freedom, using ownership types, unique pointers and partially ordered lock levels. Again, a sequential scheduler is not admissible for the model.

Some data-flow synchronization based languages and frameworks (e.g. Kahn style process networks [21], concurrent constraint programming [29], [17]) are guaranteed determinate but not safe according to our definition since they do not permit sequential schedules. Indeed they permit the possibility of deadlock. (The notion of safety is also not quite relevant since these frameworks do not support shared mutable variables.)

Synchronous programming languages like Esterel are completely deterministic, but meant for embedded targets. An Esterel program executes in clock steps and the outputs are conceptually synchronous with its inputs. It is a finite state language that is easy to verify formally. An Esterel program is susceptible to causalities. Causalities are similar to deadlocks, but can be easily detected at compile-time. The problem with synchronous models is that they do not perform well. To out knowledge, most Esterel compilers generate sequential code and not concurrent.

SHIM [16, 34] is also a deterministic concurrent programming language, but SHIM programs may deadlocks and allows only a single task to write at any phase. StreamIt [35] example is a synchronous dataflow language that provides determinism. It has simple

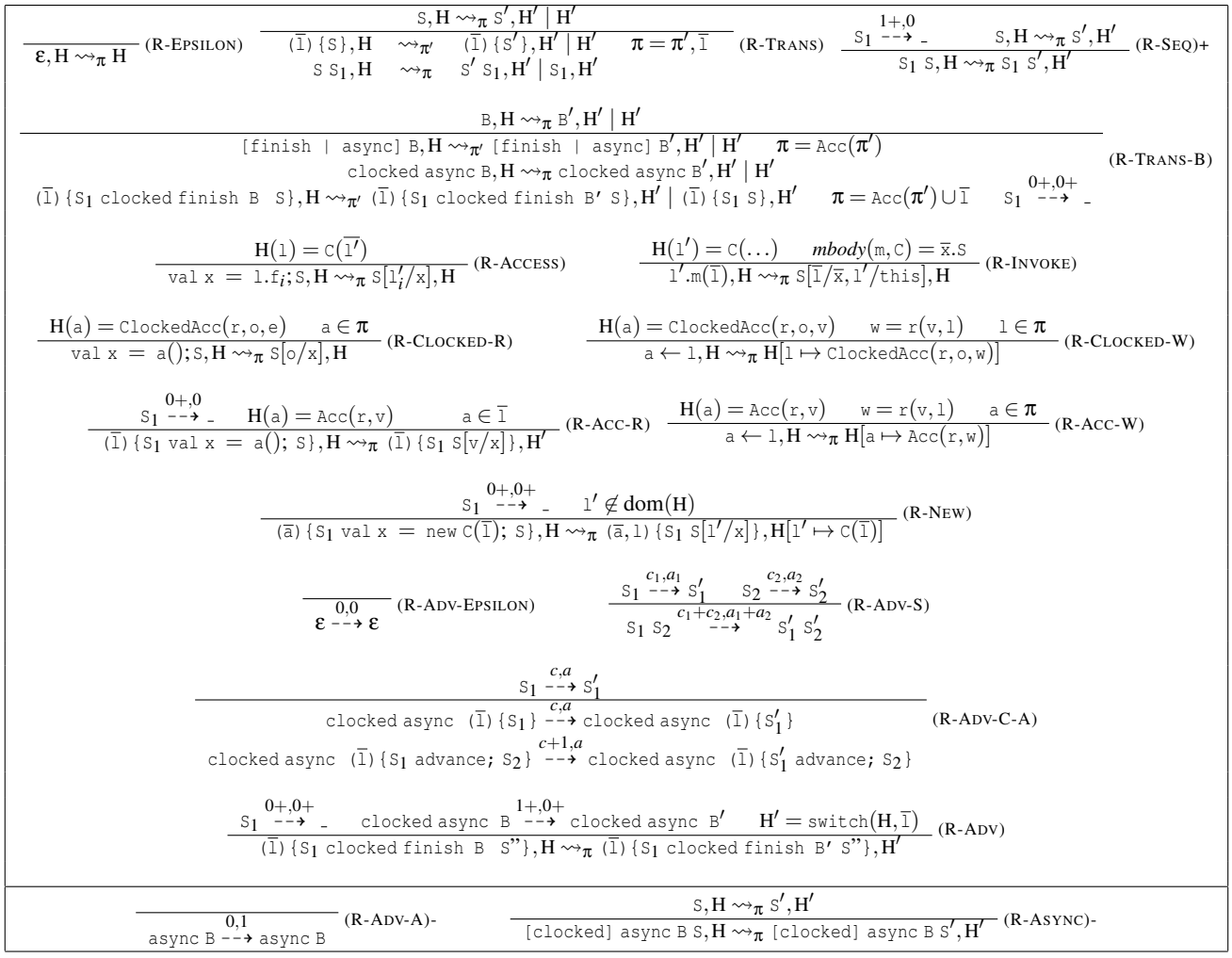

Figure 1. FX10 Reduction Rules ($S, H \rightsquigarrow_\pi S', H' \mid H'$) for the *concurrent* scheduler ((R-Seq)+ is not mandatory in the concurrent scheduler; it was added so that the sequential scheduler will progress if an `async` cannot advance.) The *sequential* scheduler is obtained by removing (R-Adv-A)- and (R-Async)-.

static verification techniques for deadlock and buffer-overflow. However, StreamIt is a strict subset of SHIM and StreamIt's design limits it to a small class of streaming applications.

In contrast, Cilk [7] is a non-deterministic language that it covers a larger class of applications. It is C based and the programmer must explicitly ask for parallelism using the *spawn* and the *sync* constructs. Cilk permits data races. Explicit techniques [12] are required for checking data races in Cilk programs.

5.2 Determinizing Tools

Determinizing run-times support coarse-grained fork-join concurrency by maintaining a different copy of memory for each activity and merging them determinately at finish points ([18], [19], [24],[2]). Safe X10 can run on such systems in principle, but does not require them. To execute Safe X10, such systems need to support fine-grained asynchrony (with some form of work-stealing or fork-joining scheduler), clocks and accumulators.

Kendo is a purely software system that deterministically multi-threads concurrent applications. Kendo [25] ensures a deterministic order of all lock acquisitions for a given program input. Kendo comes with three shortcomings. It operates completely at runtime, and there is considerable performance penalty. Secondly, if we have the sequence *lock(A); lock (B)* in one thread and *lock(B); lock(A)* in another thread, a deterministic ordering of locks may still deadlock. Thirdly, the tool operates only when shared data is protected by locks.

Software Transactional Memory (STM) [32] is an alternative to locks: a thread completes modifications to shared memory without regard for what other threads might be doing. At the end of the transaction, it validates and commits if the validation was successful, otherwise it rolls back and re-executes the transaction. STM mechanisms avoid races but do not solve the non-determinism problem.

Berger's Grace[6] is a run-time tool that is based on STM. If there is a conflict during commit, the threads are committed in a particular sequential order. The problem with Grace is that it incurs a lot of run-time overhead. Determinator[3] is another tool that allows parallel processes to execute as long as they do not share resources. If they do share resources and the accesses are unsafe, then the operating throws an exception (a page fault). Cored-Det [5], based on DMP [15] uses a deterministic token that is passed among all threads. A thread to modify a shared variable must first wait for the

token and for all threads to block on that token. DMP is hardware based. Although, deadlocks may be avoided, we believe this setting is non-distributed because it forces all threads to synchronize and therefore leads to a considerable performance penalty. In Safe X10, only threads that share a particular channel must synchronize on that channel; other threads can run independently.

Deterministic replay systems [1, 13] facilitate debugging of concurrent programs to produce repeatable behavior. They are based on record/replay systems. The system replays a specific behavior (such as thread interleaving) of a concurrent program based on records. The primary purpose of replay systems is debugging; they do not guarantee determinism. They incur a high runtime overhead and are input dependent. For every new input, a new set of records is generally maintained.

Like replay systems, Burmin and Sen [11] provide a framework for checking determinism for multi-threaded programs. Their tool does not introduce deadlocks, but their tool does not guarantee determinism because it is merely a testing tool that checks the execution trace with previously executed traces to see if the values match. Our goal is to guarantee determinism at compile time – given a program, it will generate the same output for a given input.

6. Conclusion and Future Work

6.0.1 Implementation

An early version of this design was implemented in a branch of the X10 system. Results were obtained for about twenty benchmarks, including JGF benchmarks such as IDEA, Sor, Series, RayTrace, LUFact, SparseMatMul, Geometric Mean. The speed of the programs written using accumulators and clocked types was compared to the corresponding programs written in plain X10. On the average we noticed about a 20% degradation in performance.

A new implementation is currently being worked on and we expect to have more up-to-date results for final submission. This implementation rides piggyback on the control messages exchanged to implement `finish` in order to update an `Acc` from multiple places. The rules for reads on `Acc` guarantee that intermediate accumulations can be stored at each place and do not need to be communicated to the place where the `Acc` lives until local computation has quiesced. This results in a very efficient implementation.

6.1 Adding static effects checking

Safe X10 can be substantially enriched through the addition of statically checked effects [20],[8], particularly to handle non-array-based computations. This section summarizes work currently in progress [26].

X10 already implements a very powerful dependent type system based on constraints. An object `o` is of type `T{c}` if it is of type `T` and further satisfies the constraint `c[o/self]`. Thus `Array[T]{self.rank==R}` is the type of all the arrays of `T` with rank `R`.

Types classify objects, i.e. types specify sets of objects. Therefore we can define a *location set* to be of the form `T.f` where `T` is a type and `f` is a mutable field of type `T`. It stands for the set of locations `x.f` where `x` is of type `T`. We do not need to introduce any distinct notion of regions into the language.

The key insight is that two memory locations are distinct if they either have different names or they have the same name `f` and they live in two objects `m` and `n` such that for some property `p`, `m.p != n.p` – for then it must be the case that `m != n`. Therefore we introduce enough properties into classes to ensure that we can distinguish between objects that are simultaneously being operated on. Specifically, the location sets `S{c}.f` and `S{d}.f` are disjoint if `c[x/self],d[x/self]` is not satisfiable for any `x:S`.

Methods are decorated with `@Read(L)` and `@Write(M)` annotations, where `L`,`M` are location sets. The annotation is *valid* if every read (write) of a mutable location `o.f` in the body of the method it is the case that `o.f` lies in (the set described by) `L` (`M`).

We permit location sets to be named:

```
static locs Cargos = Tree.cargo;
static locs LeftCargo(up:Tree) =
  Tree{self.up==up,self.left==true}.cargo;
```

Unlike the region system of [8] we do not need to introduce explicitly named, intensional regions, rather we can work with extensional representations of location sets (the set of all locations satisfying a certain condition). There is no need for a separate space of region names with constructors. Two location sets are disjoint if their constraints are mutually unsatisfiable, not because they are named differently. In particular, we do not need to assume that the heap is partitioned into a tree of regions.

Two key properties of the X10 type system are worth recalling. The run-time heap cannot contain cycles involving only properties. This in turn depends on the fact that the X10 type system prevents `this` from escaping during object construction[37].

We mark a property as `@ghost` to indicate that space is not allocated at run-time for this property. Therefore the value of this property is not accessible at run-time – it cannot be read and stored into variables. It may only be accessed in constraints that are statically checked. Dynamic casts cannot refer to `@ghost` properties.

We enrich the vocabulary of constraints. First, we permit existentially quantified constraints `x:T{c}` – this represents the constraint `c` in which the variable `x` of type `T` is existentially quantified. Second, we permit the *extended field selector* `e.f$i` where `e` is an expression, `f` a field name and `i` is a `UInt`.

We use an example from [8] to illustrate:

```
type
    Tree(t:Tree,l:Boolean)=Tree{self.up==t,self.left==l};
class Tree (up:Tree, left:Boolean) {
    var left:Tree(this,true);
    var right:Tree(this,false);
    var payload:Int;
    //SubTree(t) is the type of all Trees o s.t.
    // for some UInt i, o.up...up=t (i-fold iteration).
    static type
     SubTree(t:Tree)=Tree{i:UInt^self.up$i==t};
    @Safe
    def makeConstant(x:Int)
        @Write[SubTree(this)].payload
        @Read[SubTree(this)].(left,right) {
          finish {
            this.payload = x;
            if (left != null)
              async left.makeConstant(x);
            if (right != null)
              async right.makeConstant(x);
          }}}
```

In this example the fields `left`,`right` and `cargo` are mutable. It is possible to mutate a tree `p` – replace `p`'s `left` child with another tree `q`. However, `q` cannot be in `p`'s `right` subtree, because then its `up` field or `left` field would not have the right value. That is, once a `Tree` object is created it can only belong to a specific tree in a specific position.

Note that a `Tree` can be created with `null` parent and children. This is how the root is created: `new Tree(null,true)` or `new tree(null, false)`.

This is easily verified. Note that the programmer may not desire to have the fields `up` and `left` be available at run-time. These fields can be marked as `@ghost` – any attempt to access them at run-time will result in an error.

6.2 Conclusion

This paper presents a lightweight design for a safe language which permits very rich expression of concurrency idioms.

References

[1] G. Altekar and I. Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206, New York, NY, USA, 2009. ACM.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Determinator: Os support for efficient deterministic parallelism. In *9'th OSDI*, October 2010.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of Operating System Design and Implemetation (OSDI)*, 2010.

[4] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of java without data races. *SIGPLAN Not.*, 35:382–400, October 2000.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, New York, NY, USA, 2009. ACM.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[8] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. *SIGPLAN Not.*, 44:97–116, October 2009.

[9] J. Boyland. The interdependence of effects and uniqueness. In *3rd workshop on Formal Techniques for Java Programs*, 2001. URL: http://www.informatik.fernuni-hagen.de/import/pi5/workshops/ecoop2001_papers.html.

[10] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, pages 691–707, October 2010.

[11] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, pages 3–12, New York, NY, USA, 2009. ACM.

[12] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[13] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM.

[14] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. *SIGPLAN Not.*, 41:180–193, January 2006.

[15] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS*, pages 85–96. ACM, 2009.

[16] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, Sept. 2005.

[17] S. A. Edwards and O. Tardieu. Shim: A deterministic model for heterogeneous embedded systems. *Transactions on VLSI Systems*, 14:854–867, 2006.

[18] E. B. et al. Grace: Safe multithreaded programming for c/c++. In *OOPSLA*, October 2009.

[19] J. D. et al. Dmp: Deterministic shared memory multiprocessing. In *14th ASPLOS*, March 2009.

[20] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. ACM.

[21] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, Aug. 1974. North-Holland.

[22] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. *SIGPLAN Not.*, 37:246–257, May 2002.

[23] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[24] M. O. nd Jason Ansel and S. Amaransighe. Kendo: Efficient deterministic multithreading in software. In *14th ASPLOS*, March 2009.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, New York, NY, USA, 2009. ACM.

[26] Redacted. Effects as constrained types. unpublished, Aug. 2011.

[27] M. C. Rinard, C. Boyapati, C. Boyapati, and C. Boyapati. Safejava: A unified type system for safe programming. Technical report, Massachusetts Institute of Technology, 2004.

[28] M. C. Rinard and M. S. Lam. The design, implementation and evaluation of jade. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 20, 1998.

[29] V. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Award. MIT Press, Cambridge, Massachusetts, 1993.

[30] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The x10 reference manual, v2.2. unpublished, June 2011.

[31] V. Saraswat and R. Jagadeesan. *Concurrent clustered programming*, pages 353–367. Springer-Verlag, London, UK, 2005.

[32] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[33] G. L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 218–231, New York, NY, USA, 1990. ACM.

[34] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, Oct. 2006.

[35] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, Dec. 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[36] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 79–89, New York, NY, USA, 2007. ACM.

[37] Y. Zibin, D. Cunningham, I. Peshansky, and V. Saraswat. Object initialization for x10. unpublished, June 2011.