Lightweight Safe Concurrent Programming

Abstract

The safe paralleization 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 problems. 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 determinacy 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. Accumulators and clocked values are designed in such a way that they are deadlock-free and determinate even in the presence of arbitrary aliasing and patterns of concurrency.

We combine these notions with a lightweight framework for detecting disjointness of access via static effects checking.

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 can be written in this language in an "obvious" way. These patterns include histograms, all-to-all reductions, stencil computations, master/slave execution, and most of the important HPC computing idioms. 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 unrestricting aliasing in the heap. We develop the idea of an *implicit ownership domain* (first introduced in the semantics of clocks in X10 in a paper in Concur 2005) 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 determinacy and deadlock freedom 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.

There are many challenges in designing an efficient, usable, powerful, explicitly parallel imperative language that is safe. One central challenge is that

One way to get safety is through implicitly parallel languages (e.g. Jade [24],[30]). 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 [**?**] 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]

[25] 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 obtain determinacy, another idea is required. The central problem is to ensure that in the statement `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 determinate. One line of attack has been the pursuit of *effect systems* [21], [20], [9], [7]. At a broad conceptual level, 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 [7] 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 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*).

[7] 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 this approach (e.g. using the dependent-type system of X10) and extend it to cover all of X10's concurrency constructs (see Section **??**). Nevertheless the cost of developing these region assertions is not trivial. Their viability for developing large commercial-strength software systems is yet to be establishd.

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.

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.

Similarly clocked computations (barrier-based synchronization) is 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 `now` 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 `now` version, and write operation to the `next` version. This ensures that there are no read-write conflicts. There may be write-write conflicts – these must be managed by either using accumulators or an effects system. 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`.[2]

We show that clocked types can be defined in a safe way, provided that accumulators are used to resolve write-write conflicts. The only dynamic check needed is that a value of this type is 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 (clocked) accumulators. 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, SpecJBB-style communication Indeed, even some form of pipelined/systolic communication is expressible.

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 accumulators are guaranteed to be safe.

[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] 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 [**?**].

- 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 [25]. We establish that this language is safe.

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

The rest of this paper is as follows. In related-work **??**e discuss related work. In constructs **??**e present the constructs in detail, followed by examples of their use. semantics **??**resents the semantics of these constructs. We discuss implementation in implementation **??**nd finally conclude with future work.

2. Related Work

2.1 Programming Models

Determinism in parallel programming is a very active area of research. Guava [**?**] 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 [**?**] 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 [**? ?**], concurrent constraint programming [**?**], [16]) 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. 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 there are hardly any compilers that generate concurrent code off Esterel.

SHIM [15, 27] is also a deterministic concurrent programming language, but the improper use of its constructs leads to problems such as deadlocks i.e., a SHIM program may be susceptible to deadlocks. Any program written in our model is always deadlock-free. Secondly, SHIM allows only a single task to write at any phase; we allow multiply writes.

Apart from SHIM, there are a few programming models and languages that provide explicit determinism. StreamIt [28], for example is a synchronous dataflow language that provides determinism. It has simple 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 [6] 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 is definitely more expressive than D^2C. However, Cilk allows data races. Explicit techniques [12] are required for checking data races in Cilk programs.

2.2 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 ([17], [**?**], [18], [22],[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 [23] 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) [26] 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[5] 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 (determined by the order The problem with Grace is that it incurs a lot of run-time overhead. This dissertation partially solves this overhead problem by addressing the issue at compile-time and thereby reducing a considerable amount of run-time overhead.

Like Grace, 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 [4], based on DMP [14] 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 the D^2C setting, 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.

2.3 Type Systems and Verifiers

Finally, type and effect systems like DPJ [8] have been designed for deterministic parallel programming to see if memory locations overlap. Our technique is more explicit. In general, type systems require the programmer to manually annotate the program. Our model can also be implemented using annotations in existing programming languages - we in fact annotated the X10 programming language.

Martin Vechev's tool [29] finds determinacy bugs in loops that run parallel bodies. It analyzes array references and indices to ensure that there are no read-write and

3. Accumulators and Clocked Types

3.1 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 arbitarily 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.

3.2 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(z)=z`), 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 datastructures, 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`.

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. A more detailed treatment is provided in Section **??**.

Accumulators come with some dynamically-checked restrictions.

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 accumulators can be stored in heaps, since no async other the async that created the accumulator or one of its progeny can actually operate on them.

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.

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.

Accs are first-class values. There are no restrictions in storing them in data-structures, reading them, passing them as arguments to methods, returning them from methods etc.

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.

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

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 is translated to x.next()=t. That is, the value of the next field is set. 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.)

4. Programming Examples

We now demonstrate that several common idioms of concurrency and communication lie in the semantically sequential subset of X10.

4.1 Example use of `Acc`

Example 4.2 (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 4.3. `@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.

***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.

4.2 Clocked computations

Example 4.4. `@det`

```
def stencil(a:Array[Double], eps:Double, P:Int) {
 val red = Double.Reducible.max;
 val err = new ClockedAcc[Double](0.0,0.0, 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
     b.region.partition(P)) {
  //method partitions a into P pieces.
  clocked async @Write(b() | myRegion) {
   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.advanceAll( ()=> { err()=0.0; });
   }
  }
 }
 return b.finalized();
}
```

5. 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.

6. 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

6.1 Syntax

There are three predefined classes in FX10: `Object`, `Acc` (representing accumulators), and `ClockedAcc` (representing clocked accumulators). 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 clocked values (only clocked accumulators). Objects are still mutable because accumulators and clocked accumulators are mutable.

Fig. 6.2 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 {`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 `val x = e;S` evaluates `e`, assigns it to a new variable `x`, and then evaluates `S`. The scope of `x` is `S`.

Statement $p_1 \leftarrow p_2$ is legal only when p_1 is an accumulator, and it accumulates p_2 into p_1 (given the reduction operator that was supplied when the accumulator was created). Statement $p_1() = p_2$ is legal only when p_1 is a clocked value, and it writes p_2 into the next value of p_1. Expression $p()$ is legal only when `p` is an accumulator or a clocked value.

6.2 Reduction

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

The reduction relation is described in Figure 1. An *S-configuration* is of the form 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 a successfully terminated computation), or $\Diamond$ representing a computation that terminated in an error. An E-configuration is of the form 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 l,H are terminal.

Two transition relations $\rightsquigarrow_\pi$ are defined, one over S-configurations and the other over E-configurations. 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. 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 for termination, step, val, new, invoke, and access 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` S_1. Further, each block records its set of synchronous capabilities. When descending into the body, the block's own capabiliites are added to those obtained from the environment.

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.) Given a set of registered objects $\overline{l}$ in a heap H, we define $\mathrm{Acc}(\overline{l})$ to return only the accumulators in $\overline{l}$.

The rule (R-ACCESS) ensures that the field is initialized before it is read (f_i is contained in $\overline{f}$). The rule (R-ACC-A-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`.

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 (if not, an erorr is thrown).

The rule (R-ADVANCE) permits a `clocked finish` to advance only if all the top-level `clocked asyncs` in the scope of the `clocked finish` and before an `advance` are stuck at an `advance`. A `clocked finish` can also advance if all the the top-level `clocked asyncs` in its scope are stuck. In this case, the statement in the body of the `clocked finish` has terminated, and left behind only possibly clocked `async`. This rule corresponds to the notion that the body of a `clocked finish` deregisters itself from the clock on local termination. Note that in both these rules un-clocked `aync` may exist in the scope of the `clocked finish`; they do not come in the way of the `clocked finish` advancing.

6.3 Results

7. Conclusion and Future Work

7.1 Adding static effects checking

The richness of the existing language can be substanitally enriched through the addition of statically checked effects [19],[7].

We discuss briefly how these ideas can be integrated into Safe X10, thereby substantially enriching its power. This summarizes work currently in progress.

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]{this.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 [7] 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[**?**].

We mark a property as `@ersatz` 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 `@ersatz` properties.

We enrich the vocabulary of constraints. First, we permit existentially quantified constraints `x:T`$\hat{c}$ – 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 [7] 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)`.

In checking the validity of the effect annotation on `makeConstant` the compiler must check that the effects of the body are contained in the declared effects of the method. This boils down to checking the subtyping relations. e.g. for the `@Write` annotation it must check:

```
SubTreeTree
Tree{i:UInt^(self.up$i.up==this,self.up$i.left==true}
    <: Tree{i:UInt^(self.up$i==this}
Tree{i:UInt^(self.up$i.up==this,self.up$i.left==false}
    <: Tree{i:UInt^(self.up$i==this}
Tree{self==this} <: Tree{i:UInt^(self.up$i==this)}
```

This is easily verified. The notion of "distinctions from the left" and "distinctions from the right" of [7] are not needed. These arise naturally through the use of constraints (distinct access paths, vs distinct fields).

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.

Constraints such as "this array must point to distinct objects" can also be naturally represented in the dependent type system. For more details please see [**?**].

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] 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] 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.

[5] 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.

[6] 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.

[7] 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.

[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. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, New York, NY, USA, 2009. ACM.

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

[15] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] M. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In R. Cousot and M. Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 455–471. Springer Berlin / Heidelberg, 2011.

[30] 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.

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

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. 1.

$$\frac{}{\varepsilon,\mathrm{H} \leadsto_\pi \mathrm{H}}\ \text{(R-EPSILON)} \qquad \frac{\mathtt{S},\mathrm{H} \leadsto_\pi \mathtt{S}',\mathrm{H}' \mid \mathrm{H}' \mid \Diamond}{\begin{array}{lll}(\overline{\mathtt{l}})\ \{\mathtt{S}\},\mathrm{H} & \leadsto_{\pi'} & (\overline{\mathtt{l}})\ \{\mathtt{S}'\},\mathrm{H}' \mid \mathrm{H}' \mid \Diamond \qquad \pi = \pi',\overline{\mathtt{l}} \\ \mathtt{S\ S_1},\mathrm{H} & \leadsto_\pi & \mathtt{S'\ S_1},\mathrm{H}' \mid \mathtt{S_1},\mathrm{H}' \mid \Diamond\end{array}}\ \text{(R-TRANS)}$$

$$\frac{\mathtt{B},\mathrm{H} \leadsto_\pi \mathtt{B}',\mathrm{H}' \mid \mathrm{H}' \mid \Diamond}{\begin{array}{c}\mathtt{async\ B},\mathrm{H} \leadsto_{\pi'} \mathtt{async\ B}',\mathrm{H}' \mid \mathrm{H}' \mid \Diamond \qquad \pi = \mathtt{Acc}(\pi') \\ \mathtt{finish\ B},\mathrm{H} \leadsto_{\pi'} \mathtt{finish\ B}',\mathrm{H}' \mid \mathrm{H}' \mid \Diamond \qquad \pi = \mathtt{Acc}(\pi') \\ \mathtt{clocked\ async\ B},\mathrm{H} \leadsto_\pi \mathtt{clocked\ async\ B}',\mathrm{H}' \mid \mathrm{H}' \mid \Diamond \\ (\overline{\mathtt{l}})\ \{\mathtt{S_1\ clocked\ finish\ B\ \ S}\},\mathrm{H} \leadsto_{\pi'} (\overline{\mathtt{l}})\ \{\mathtt{S_1\ clocked\ finish\ B'\ S}\},\mathrm{H}' \mid (\overline{\mathtt{l}})\ \{\mathtt{S_1\ S}\},\mathrm{H}' \mid \Diamond \qquad \pi = \mathtt{Acc}(\pi') \cup \overline{\mathtt{l}} \qquad \mathtt{S_1} \overset{0+,0+}{\dashrightarrow} \mathtt{S}_1'\end{array}}\ \text{(R-TRANS-B)}$$

$$\frac{\mathtt{S},\mathrm{H} \leadsto_\pi \mathtt{S}',\mathrm{H}'}{\mathtt{[clocked]\ async\ B\ S},\mathrm{H} \leadsto_\pi \mathtt{[clocked]\ async\ B\ S}',\mathrm{H}'}\ \text{(R-ASYNC)-}$$

$$\frac{\mathtt{S_1} \overset{1+,0}{\dashrightarrow} \mathtt{S}_1' \qquad \mathtt{S},\mathrm{H} \leadsto_\pi \mathtt{S}',\mathrm{H}'}{\mathtt{S_1\ S},\mathrm{H} \leadsto_\pi \mathtt{S_1\ S}',\mathrm{H}'}\ \text{(R-SEQUENTIAL)+}$$

$$\frac{\mathrm{H}(\mathtt{l}') = \mathtt{C}(\ldots) \qquad \mathit{mbody}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{x}}.\mathtt{S}}{\mathtt{l}'.\mathtt{m}(\overline{\mathtt{l}}),\mathrm{H} \leadsto_\pi \mathtt{S}[\overline{\mathtt{l}}/\overline{\mathtt{x}},\mathtt{l}'/\mathtt{this}],\mathrm{H}}\ \text{(R-INVOKE)}$$

$$\frac{\mathrm{H}(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}'})}{\mathtt{val\ x\ =\ l.f}_i;\mathtt{S},\mathrm{H} \leadsto_\pi \mathtt{S}[\mathtt{l}_i'/\mathtt{x}],\mathrm{H}}\ \text{(R-ACCESS)}$$

$$\frac{\mathrm{H}(\mathtt{l}) = \mathtt{ClockedAcc}(\mathtt{r},\mathtt{o},\mathtt{e}) \qquad \mathtt{l} \in \pi \mid \mathtt{l} \notin \pi}{\mathtt{l}() = \mathtt{l}',\mathrm{H} \leadsto_\pi \mathrm{H}[\mathtt{l} \mapsto \mathtt{ClockedAcc}(\mathtt{r},\mathtt{o},\mathtt{r}(\mathtt{e},\mathtt{l}'))] \mid \Diamond}\ \text{(R-CLOCKED-A)} \qquad \frac{\mathrm{H}(\mathtt{l}) = \mathtt{ClockedAcc}(\mathtt{r},\mathtt{o},\mathtt{e}) \qquad \mathtt{l} \in \pi \mid \mathtt{l} \notin \pi}{\mathtt{val\ x\ =\ l()};\mathtt{S},\mathrm{H} \leadsto_\pi \mathtt{S}[\mathtt{o}/\mathtt{x}],\mathrm{H} \mid \Diamond}\ \text{(R-CLOCKED-R)}$$

$$\frac{\mathtt{S_1} \overset{0+,0}{\dashrightarrow} \mathtt{S}_1' \qquad \mathrm{H}(\mathtt{a}) = \mathtt{Acc}(\mathtt{r},\mathtt{v}) \qquad \mathtt{a} \in \overline{\mathtt{l}} \mid \mathtt{a} \notin \overline{\mathtt{l}}}{(\overline{\mathtt{l}})\ \{\mathtt{S_1\ val\ x\ =\ a();\ S}\},\mathrm{H} \leadsto_\pi (\overline{\mathtt{l}})\ \{\mathtt{S_1\ S}[\mathtt{v}/\mathtt{x}]\},\mathrm{H}' \mid \Diamond}\ \text{(R-ACC-R)} \qquad \frac{\mathrm{H}(\mathtt{a}) = \mathtt{Acc}(\mathtt{r},\mathtt{v}) \qquad \mathtt{w} = \mathtt{r}(\mathtt{v},\mathtt{p}) \qquad \mathtt{a} \in \pi \mid \mathtt{a} \notin \pi}{\mathtt{a} \leftarrow \mathtt{p},\mathrm{H} \leadsto_\pi \mathrm{H}[\mathtt{a} \mapsto \mathtt{Acc}(\mathtt{r},\mathtt{w})] \mid \Diamond}\ \text{(R-ACC-W)}$$

$$\frac{\mathtt{S_1} \overset{0+,0+}{\dashrightarrow} \mathtt{S}_1' \qquad \mathtt{l}' \notin \mathrm{dom}(\mathrm{H})}{(\overline{\mathtt{m}})\ \{\mathtt{S_1\ val\ x\ =\ new\ C}(\overline{\mathtt{l}});\ \mathtt{S}\},\mathrm{H} \leadsto_\pi (\overline{\mathtt{m}},\mathtt{l})\ \{\mathtt{S_1\ S}[\mathtt{l}'/\mathtt{x}]\},\mathrm{H}[\mathtt{l}' \mapsto \mathtt{C}(\overline{\mathtt{l}})]}\ \text{(R-NEW)}$$

$$\frac{}{\varepsilon \overset{0,0}{\dashrightarrow} \varepsilon}\ \text{(R-ADV-EPSILON)} \qquad \frac{}{\mathtt{async\ B} \overset{0,1}{\dashrightarrow} \mathtt{async\ B}}\ \text{(R-ADV-A)-}$$

$$\frac{\mathtt{S_1} \overset{c,a}{\dashrightarrow} \mathtt{S}_1'}{\begin{array}{c}\mathtt{clocked\ async\ \ (\overline{l})\ \{S_1\}} \overset{c,a}{\dashrightarrow} \mathtt{clocked\ async\ \ (\overline{l})\ \{S_1'\}} \\ \mathtt{clocked\ async\ \ (\overline{l})\ \{S_1\ advance;\ S_2\}} \overset{c+1,a}{\dashrightarrow} \mathtt{clocked\ async\ \ (\overline{l})\ \{S_1'\ advance;\ S_2\}}\end{array}}\ \text{(R-ADV-C-A)}$$

$$\frac{\mathtt{S_1} \overset{c_1,a_1}{\dashrightarrow} \mathtt{S}_1' \qquad \mathtt{S_2} \overset{c_2,a_2}{\dashrightarrow} \mathtt{S}_2'}{\mathtt{S_1\ S_2} \overset{c_1+c_2,a_1+a_2}{\dashrightarrow} \mathtt{S}_1'\ \mathtt{S}_2'}\ \text{(R-ADV-S)}$$

$$\frac{\mathtt{S_1} \overset{0+,0+}{\dashrightarrow} \mathtt{S_1} \qquad \mathtt{clocked\ async\ \ B} \overset{1+,0+}{\dashrightarrow} \mathtt{clocked\ async\ \ B}' \qquad H' = \mathtt{switch}(H,\overline{\mathtt{l}})}{(\overline{\mathtt{l}})\ \{\mathtt{S_1\ clocked\ finish\ \ B\ \ S''}\},\mathrm{H} \leadsto_\pi (\overline{\mathtt{l}})\ \{\mathtt{S_1\ clocked\ finish\ B'\ S''}\},\mathrm{H}'}\ \text{(R-ADV)}$$

Figure 1. FX10 Reduction Rules ($\mathtt{S},\mathrm{H} \leadsto_\pi \mathtt{S}',\mathrm{H}' \mid \mathrm{H}' \mid \Diamond$) for the *concurrent* scheduler ($(\ldots)+$ is not mandatory). The *sequential* scheduler is obtained by removing $(\ldots)$