A Theory of Memory Models

Vijay Saraswat [*] Radha Jagadeesan [†] Maged Michael [‡] Christoph von Praun [§]

Abstract

A *memory model* for a concurrent imperative programming language specifies which writes to shared variables may be seen by reads performed by other threads. We present a simple mathematical framework for relaxed memory models for programming languages. To instantiate this framework for a specific language, the designer must choose the notion of *atomic steps* supported by the language (e.g. 32-bit reads and writes) and specify how a composite step may be broken into a sequence of atomic steps (the *decomposition rule*). This rule determines which sequence of intermediate writes (if any) are visible to concurrent reads by other threads. Different choices of the rule lead to models which permit a read to return *any* value if there is a concurrent write (race), or models which satisfy a "No Thin Air Read" property. The former is suitable for languages such as C++ (programs with races have undefined behavior), and the latter for Java. Other intermediate models are possible, useful and interesting.

We establish that all models in the framework satisfy the *Fundamental Property* of relaxed memory models: programs whose sequentially consistent(SC) executions have no races must have have only SC executions. We show how to define synchronization constructs (such as volatiles, of various kinds) in the framework, and discuss the causality test cases from the Java Memory Model.

1. Introduction

Memory models address a central question of imperative concurrency: When can a write done by one thread be read by another?

Leslie Lamport provided a simple answer in [Lam79]. Assume the state of the memory can be described by an assignment of values to variables. Assume that exactly one thread is permitted to perform exactly one read or write operation in a single step. Then the possible executions of the program are given by all possible interleavings of the steps of the threads making up the program. This notion of execution is called *Sequential Consistency (SC)*.

Unfortunately, SC is not consistent with a wide array of compiler optimizations geared towards optimizing the performance of single-threaded code. Such optimizations often work by rearranging the code of a single thread while guaranteeing that its *input/output (i/o) behavior* is unchanged. For any piece of sequential code s, let us define its *i/o function* $io(s)$ to be the function from total stores (mappings that assign a value to every variable) to total stores given by executing s in the input store and returning the final store. Consider, for instance, the program P, for `r`, `x` and `y` distinct variables: `x=1; r=y;`. An implementation may replace this code by P': `r=y; x=1` since $io(P) = io(P')$. However, under SC these two code fragments are not identical. Consider running it in parallel with Q: `r0 = x; if (r0 ==1) y=1`. Assume execution is initiated in a store in which `x=0, y=0`. Now $P \mid Q$ may result in `r=1`, whereas $P' \mid Q$ will never do so. Thus, SC makes it difficult for multiple threads to perform multiple operations on memory simultaneously, contrary to what is done by modern architectures.

It should be noted that Shasha and Snir [SS88] recognized this problem and proposed solutions involving extra computational overhead (e.g. the use of memory barriers/fences). There has been more recent work [SFW$^+$05] on compiler analyses to reduce or eliminate the overhead of implementing SC. At this point we cannot definitively conclude that the overhead can be eliminated for a large class of programs. Therefore the need to define memory models is real.

1.1 Race-free programs

An important observation underlies nearly all research in this area. Consider again the program $P' \mid Q$ above. Let us say that steps executed by a program are related with a transtive, irreflexive partial order, the *happens-before* (*hb*) order [Lam79].

> An order is a binary relation, say, $\leq$. It is transitive if $a \leq b$ and $b \leq c$ implies $a \leq c$. It is irreflexive if it is not the case for any a that $a \leq a$. It is partial if it is not necessarily the case that for any two elements a and b either $a \leq b$ or $b \leq a$. If one views such an order as a directed graph with elements as nodes and an edge from a to b if $a \leq b$, then the graph will satisfy the property that it is a forest of dags.

One should interpret p *hb* q as saying that the step p must happen "before" the step q in any execution; i.e. q must observe the store in a state in which p has been performed. For instance, it is reasonable to require that all the steps taken by a single thread are totally ordered by *hb*, and synchronization operations (e.g. lock/unlock) must be used to (dynamically) introduce *hb* edges between steps of one thread and steps of another. Now since $P' \mid Q$ does not contain any synchronization operation, it has a *data race*: a thread (Q) has a step s (`r0=x`) that reads a variable (`x`) that another thread (P') writes in a step t (`x=1`) without there being an *hb*-edge from t to s.[1] If a program has no races then a thread T_1 does not read the value of a variable written into by another thread T_2 (without using a synchronization operation). Therefore T_1 will be insensitive to code reordering in T_2. Hence one can have one's

[*] IBM T.J. Watson Research Center, vijay@saraswat.org

[†] DePaul Univ., rjagadeesan@cs.depaul.edu

[‡] IBM T.J. Watson Research Center,magedm@us.ibm.com

[§] IBM T.J. Watson Research Center,praun@us.ibm.com

[1] Two steps are in a race if both read or write the same variable x, at least one of them writes to x and the steps are not ordered by *hb*.

cake (SC semantics) and eat it too (good performance). Therefore it seems reasonable to require that *programs whose SC executions have no races must have only SC executions*. We shall call this property the *Fundamental Property*.

This raises the question: Who is responsible for ensuring that a program is race free ... the implementation[2] or the user?

It is plausible that the implementation should have this responsibility. Race analysis is a difficult technical problem . Much as languages have been designed that do not suffer from aliasing problems, it may be possible to design languages without race conditions. In some cases it may be permissible to incur the overhead of runtime detection of races. This approach is being pursued by some researchers. The general drawback is that it is hard to design static conditions that are general enough to recognize that arbitrary clever programs are race-free.

Indeed, it is often the case that a programmer – aware of the designed control flow of the program – can establish that a particular program is race-free based on global analysis. The use of `finish`/`foreach` operations often results in such programs in X10 [SJ05,CDG+05].[3]

EXAMPLE 1 (Red-black iteration) Consider a canonical dense-array computation, red-black iteration:

```
for (point p: [1:NUM_ITERS]) {
 finish foreach (point i : r)
    red[i] = average(black, i, stencil[i]);
 finish foreach (point i : r)
    black[i] = average(red, i, stencil[i]);
```

Here the programmer has arranged for `red` and `black` to reference different (non-overlapped) arrays, and `r` is a potentially large region (e.g. millions of elements). For every point `i` in the region `stencil[i]` specifies a region relative to `i`. It is desired that this program result in the parallel evaluation of the average value over the given region, for every point in `r`, with the resulting value being written into the corresponding point in `red`. Notice that the logic of the program ensures there are no concurrent reads/writes to the same location. (The `finish` operation introduces a global barrier which enforces the desired sequencing between reads and writes to the same location.) Once all activities have finished the result of the writes are available to all activities spawned subsequently by the main thread.

To execute the X10 program efficiently, the implementation must not be forced to perform fine-grained synchronization actions (e.g. memory fences) at each `red`/`black` read/write. Coarse-level synchronization mechanisms to ensure race-freedom have already been put in place by the programmer. □

1.2 Requirements for Relaxed Memory Models

Therefore it seems plausible that the programmer should shoulder the responsibility of establishing the global property that synchronization-free access to shared variables will not lead to race conditions. In return, the implementation should guarantee performance: it should be able to perform *all single thread optimizations* as long as they are consistent with explicit synchronization operations introduced by the programmer (if any). That is, the language should specify – and the implementation should realize – as *weak* semantics as possible for concurrent, unsynchronized read/writes to the same location (performing as many code reorderings as possible). Memory barriers should be introduced only as required by the semantics of synchronization operations in the language.

How weak is "weak"?

Fundamental Property. For programs **without** races, the Fundamental Property places a lower bound for programs without races. This property appears to be a reasonable "firewall": most programmers may program in a world in which they write complete, race-free programs and only worry about SC executions.

We shall require this property for all relaxed memory models. However stronger versions of this property may be more relevant to practice, e.g. versions which permit libraries with races in them but permit the programmer to firewall these races in a way that cannot interfere with clients of the library. We leave the topic of stronger versions of the Fundamental Property to future work.

No Thin Air Reads?? For programs **with** races, different answers are possible. Consider a language such as C++ in which programs with races are considered to be erroneous and their behavior is undefined. In such a case all transformations should be permitted as long as only programs with races can distinguish between them. For instance, it should be possible to replace any write `x=y` with the i/o equivalent `x=42;x=y`. Only a program with a race would be able to see the "out of thin air" write `42`.

Such a transformation is not as unreasonable as it may appear. For instance a vectorizing compiler may wish to pack multiple variables `x,y,z,u` into two long words and use vector instructions to optimize execution. The code `x=1;y=1;z=1;u=0` may be implemented with the two-instruction sequence `x,y,z,u=1,1,1,1;u=0`. Now the implementation has introduced a Thin Air Write `u=1` which can be detected by a program with races.

On the other extreme, in a language such as Java, which satisfy the property that certain data types, such as object references, behave like capabilities. A piece of code can obtain a reference *r* to an object only if it creates the object or it reads a memory location containing that reference. The integrity of large applications written in such languages relies on the property that references to objects can be "closely held", i.e. held only by a certain collection of programmer-specified objects. A semantics which permits Thin Air Reads would permit an attacker to introduce code into the system (e.g. with an applet) which may gain access to such a closely held object via some sequence of seemingly innocuous transformations.

A litmus test for "No Thin Air Reads" is the following test case from [Pug04]. (For the convenience of the reader we indicate with each example the corresponding test number in [Pug04] using the (TC xx) notation. For now, we use an informal notation for programs. We formalize the syntax in Section 3.)

EXAMPLE 2 (TC 4) See also [MPA05, Fig 2]. Consider the program

```
x=0;y=0;( r1=x; y=r1 | r2=y; x=r2)
```

Such a program may not exhibit the behavior `r1==r2==1`; values are not allowed to materialize out of thin air. □

A related case is exemplified by the following variant of TC 2.

EXAMPLE 3 Consider the program

```
x=0;y=0;(r1=x; r2=x; y=(r1==r2)?1; | r3=y;x=r3)
```

Such a program should not exhibit the observation `r1=0,r2=1,r3=1`, since the only justification for `r3=1` appears to require `r1==r2`. □

Note though that it is not difficult for a compiler to transform the program above so that the behavior *is* possible. For instance,

[2] Throughout this paper, when we say "implementation" we mean the compiler/run-time system/architecture/hardware – that is, all elements of the language implementation.

[3] We emphasize that the ideas in this paper are applicable to all shared memory concurrent languages. We use X10 simply for illustration.

it replaces the first thread with the i/o equivalent code sequence `y=1;r1=x;r2=x`. An SC execution yields this result, e.g. via:

```
x=0;y=0;y=1;r1=x;r3=y;x=r3;r2=x;x=1
```

In summary, we shall require that a framework for memory models must be flexible enough to permit the formulation of memory models that answer these test cases differently. Such a framework permits programming language designers to choose a variation appropriate for their language.

Inlinability. The Red-Black interation example above illustrates another important requirement for the memory model for X10[SJ05, CDG+05] like languages that encourage the use of asynchrony. Any particular X10 implementation is likely to have fewer hardware threads than the number of activities spawned by the computation. Therefore it is necessary for the X10 implementation to ensure that activities are aggregated. For instance, the X10 compiler should be free to chunk the red-black iterations above with arbitrary granularity, depending on the number of hardware threads available. Such chunking should not impose any additional runtime cost because of extra synchronization. Therefore we require that the memory model support the ability to "inline" activities, wherever this does not cause deadlock; e.g. typically activities executing a potentially blocking `when` operation, or a clock `next`, will not be inlined.

Usability. Programmers need to use the memory model to understand all possible behaviors of their programs. Therefore memory models should be developed using concepts and terminology familiar to programmers. Programmers understand programs: hence, as far as possible, a memory model should be presented in terms of a few simple permitted transformations of programs that generate permitted behaviors. A programmer should be able to calculate all possible behaviors of a program by systematically applying these transformations.

Requirements summary. We may now summarize the memory model requirements for X10 like languages. These requirements are based on the fundamental assumption that the responsibility for ensuring that a program is correctly synchronized lies with the programmer. The nemory model framework must:

1. Ensure that for every model satisfies the Fundamental Property.
2. Be flexible enough to permit different formulations of the "No Thin Air Reads" principle.
3. Permit unrestricted use of single-thread optimizations (e.g. code reordering), subject to the two previous conditions.
4. Require the introduction of explicit memory synchronization operations, such as fences, only as necessary to implement explicit synchronization operations in the language (e.g. `atomic`, `when`, `clocks`).
5. Permit the compiler to perform whole program analysis and replace an arbitrary single-thread code fragment C with another fragment that is equivalent to it under constraints on the value of data variables that are true at (the beginning of) C in all SC executions of the program.
6. Specify a few rules that can be used by the programmer to systematically enumerate all possible execution sequences for a given program (even if it has races).

It is desirable that the following programming methodology be supported by the memory model:

- Most programmers should use explicit synchronization operations (`atomic`, `when`) to reliably communicate values between activities via shared variables.
- For better performance, programmers may use unsynchronized access to variables provided they ensure the global property that there are no data races involving these variables. They may then reason about their program using sequential consistency.
- If the program contains data races, the memory model must specify a small set of rules that may be used by the programmer/compiler to reason about programs.

1.3 The basic model

We briefly present the central ideas underlying the memory model, deferring formal details to the main body of the paper.

The central idea behind the models presented in this paper is to formalize sequential execution through the notion of a *step*. Intuitively, a step is a kind of sequential function which reads and writes variables in a store, and performs computations on them. Each programming language will come equipped with its own notion of primitive, indivisible (atomic) steps (e.g. read or write a 32-bit variable), and with a translation function which maps programs in the language to sequences of such primitive steps. Steps should be closed under sequential composition – if $s_1, \ldots, s_n$ are steps, then $s_1; \ldots; s_n$ should also be a (composite) step – since sequential programs will execute sequences of such steps.

These intuitions may be developed precisely as follows. Let us say that a partial store is an assignment of values to variables; such a store is *total* if it specifies values for all variables. A crucial move is to consider a step to be a *partial write function*. For a sequence of statements s, $pw(s)$ is the partial function from partial stores to partial stores which is defined on an input store d only if d specifies values for all the variables that are read by s, and it maps such a d to the set of *writes* produced by running s. Thus a step carries more information than just the i/o function – intuitively, it records the set of variables read as well as the set of variables actually written by the program. For instance the behavior of the program `skip; x=x;` is different from `skip` (even though both have the same i/o function), since the former can cause a race whereas the latter can not. Similarly the program `x=y;x=z` is different from the program `x=z` (even though both have the same i/o function) since the former may be involved in a race with `y` but the latter can not. Partial write functions make these extra distinctions while being able to recover the i/o function, if needed.

With such a view of sequential execution in hand, the notion of concurrent execution is easy to define: it is a collection (multiset) of steps with two bits of additional structure. First there must be a partial order on steps arising from sequentiality of steps executed by the same thread (the "happens before" order). Second, there must be a way to reflect *links* that record which step f was used to answer the read of a variable x by a step g. The links must satisfy a consistency condition with the happens before relation, namely, if a link connects a step f to a step g on a variable x, then either f and g are unordered or there is no other step between f and g (in the *hb*-order) which writes on x. (This condition is called the hb-*consistency condition*.)

Transformations. A *process* is taken to be a set of pomsets (with links), closed under a certain set of simple transformations. All models in the RAO (Relaxed Atomic + Ordering) family are equipped with the transformations *improvement* (IM), *composition* (CO), *link*(LI), *propagation* (PR), and *augmentation* (AU). Additionally, each model has a *decomposition transformation*, DX, which refines the "weakest" decomposition transformation, DL.

IM permits a step f to be replaced by a step g if $io(f) = io(g)$, and g reads and writes fewer variables than f, while respecting all incoming and outgoing links. IM permits extra reads and writes to be dropped (e.g. `x,y=x,2` to be replaced by `y=2`). CO permits

two successive steps $g;h$ to be replaced by $f = g \circ h$, as long as incoming and outgoing links and *hb* edges are respected.

DL permits a step f to be replaced by a pair of atomic steps $g;h$ as long as $f = g \circ h$ and incoming and outgoing links are respected. While this restriction is strong enough to guarantee that no new races are introduced, it permits the replacement of `x=y` by `x=42;x=y` and hence invalidates Examples 2 and 3. Stronger decomposition rules are also permitted, however they must all strengthen DL (i.e. impose extra conditions). For instance, DR requires additionally that g and h must write any variable at most once (but may read a variable more than once). Similarly DW requires that g and h read any variable at most once (but may write a variable multiple times). DO imposes both conditions. DW validates 2 and 3; DR validates 3 but not 2; DO validates both. Other decomposition rules are also possible.

The decomposition rule for a programming language specifies the intermediate reads and writes that can be performed when decomposing a composite step. The requirement that decomposition rules strengthen DL ensures that they cannot introduce new races.

AU is the only transformation that changes the *hb* relation between existing steps. An *hb*-edge can be added between two steps provided that the result of the transformation is a valid process. This transformation is not supported by the Java Memory Model described in [MPA05].

PR is a generic "whole program" transformation. It permits a step f to be replaced by a step g provided that f and g are equivalent in all stores that satisfy a condition c, and it is the case that all SC executions of the program force the condition c to be true before f is executed. PR permits whole program analysis to be factored into the model.

LI is a restricted form of composition that permits a step to "read" the information in another *hb*-unordered steps by introducing a *link*. Let f and g be two *hb*-unordered steps. LI is parameterized by a non-empty set of variables W. It links f and g so that information produced by f on W is used to update the input store into g. Thus g "sees" the writes on W performed by f. The information produced by f is not communicated by g to the output. Note that LI and AU interact: AU may not add an edge that violates the conditions of any existing link.

CO, DX, IM, AUand PRare compatible with a totally-ordered notion of memory — memory is a global set of locations from which every read fetches the current value and every write modifies the current value. However, LI permits the same thread to see two different writes if it performs two different reads of the same variable in sequence, even if no other thread has taken any action in the meantime. Thus, LI, when it links a read to an unordered write, is *not* compatible with such a "Totally Ordered Memory" principle.

These six transformation rules provide for a rich range of behaviors. They can be used to obtain the effect of interleaving "execution" (e.g. DX,LI) with "compilation" (application of PR, requiring whole program analysis). Therefore this framework is particularly appropriate for JITted languages which permit such interleaving of compilation and execution.

The RAO model does not support the notion of "a central store" shared by all threads. Therefore the notion of execution must now explicitly take the partial order into account. But this is easy. Say that a step s is *completed* if all its reads have been answered and so it is "fully-defined." (We formalize this in Section 2.) Say that a process is completed if all its steps are completed. An *execution* of an AO process P is just any completed process P' obtained by P by repeated application of the RAO transformations. (The introduction of synchronization operations may introduce additional conditions on an execution, for instance, all acquire and release steps on the same lock are totally ordered.)

The RAO transformations preserve the fundamental property – programs without race conditions in SC executions have only SC executions – and also possess a "Thin Air Read" property". By design, they permit a large class of transformations typically implemented by modern day language implementations. In particular, we show that they provide a satisfactory account of all the test cases in [Pug04].

On this basis various synchronization operations can be defined. We introduce the idea of *raw* and *volatile* variables. Raw variables do not possess any special synchronization properties. Multiple threads may read and write raw variables simultaneously, and a write is visible to read if it is *hb*-consistent (as discussed above). Thus raw variables correspond to the variables we have discussed above. Volatile variables introduce synchronization conditions. In this paper, we consider three variants of volatility, one introduced in JLS 2 (the weakest), DX-restricted volatility, and JLS 3 volatility. Other notions of volatility, and other synchronization operations such as atomics, locks etc may be defined on top of RAO. They will be dealt with in subsequent papers.

2. RAO Model

2.1 Preliminaries

First some simple preliminaries to fix intuitions.

Stores. By a *partial function* from a domain D to a range R we shall mean a function that is defined from some subset of D, $dom(f)$, into R. The *restriction* of a partial function f to a set V, $f \downarrow V$, is f restricted to the domain $dom(s) \cap V$.

We fix an infinite set of variables V and a set of values L. A *partial store* d is a partial map from V to L, a *total* store is one whose domain is V. We designate the set of all partial stores by *Store*, and the set of all total stores by *TStore*. We treat a store isomorphically as a set of bindings, $\{x_0 = v_0, x_1 = v_1, \ldots\}$.

The *union* $d_0 \cup d_1$ of two stores d_0 and d_1 (with disjoint domains) is their union when viewed as a set of bindings. Since two stores may have conflicting information, their *asymmetric union* $c[d]$ (read as: *c updated by d*) is quite important and is defined as the set of bindings in d together with the bindings from c for those variables not bound by d.

We define a binary relation on stores $c \leq d$ (read as: *d extends c*) to hold iff $d[c] = d$. It is easy to see that $\leq$ is a partial order. Note that for distinct stores d, d', $d \leq d'$ implies that $dom(d)$ is strictly contained in $dom(d')$.[4]

Functions on stores. Define a partial order on partial functions over stores by: $f \leq g$ if $dom(f) \subseteq dom(g)$ and $f(c) \leq g(c)$ for all $c \in dom(f)$. The notion of *monotonicity* of such functions is standard. f is *monotone* if $d \in dom(f), e \geq d$ implies $e \in dom(f)$ and $f(e) \geq f(d)$. For any function f on stores, we define its *transition function* $f^\sharp$ by: $f^\sharp(c) = c[f(c)]$. Unlike f, $f^\sharp$ "flows" the input through to the output.

A function f is *complete* if it is defined for every total store.

2.2 Modelling single-threaded code

The fundamental intuition underlying the models is that a piece of sequential code should be modelled as a *step*, i.e. a *function* from stores to stores. The fundamental attraction of a function in the current context is that two pieces of code that have the same single thread behavior specify the same function. Thus functions offer a convenient abstraction that respects single thread optimizations.

[4] That is, the only way to rise "strictly higher" in the $\leq$ ordering is to be defined for more variables. This is a characteristic of "concrete domains", domains in which there is no partial order on values, hence no notion of improving values.

What kind of functions? The simplest idea would be to consider functions over total stores (stores that assign a value to every variable). The function maps an input store to the (ordered) output store that results from running the given sequential program s. Let us call such a function an *i/o function for* s, $io(s)$. $io(s)$ corresponds to a sequentially consistent execution of s: computation is initiated in a store in which every variable has a value, the execution of the statement reads the value of the variable from the store whenever it is required, and updates the store immediately, indivisibly and atomically whenever it executes an assignment. The result of execution is a total store.

The use of $io(s)$ for relaxed memory models has two main drawbacks. First, we wish to distinguish between the piece of code `skip;` and `skip; x=x` since the latter may cause a race condition on `x` while the former will not. However, both yield the same i/o function. We say that the i/o function is *oblivious to extra writes*. So we must use some other kind of function.

One way to distinguish them is to use functions that record for each total input store d the **actual writes** w produced by running the sequential program s. Let us call such a function the *write function,* $w(s)$ for s. Thus $w(\texttt{skip})$ maps each input store to $\{\}$, whereas $w(\texttt{skip};\texttt{x}=\texttt{x})$ maps each input store d to $\{x = d(x)\}$. Note that $io(s)$ is just $w(s)^\sharp$.

Second, we wish to record which variables in the input store need to be read by the program s in order to produce a write. For instance $w(\texttt{x}=\texttt{z};\texttt{x}=\texttt{y}) = w(\texttt{x}=\texttt{y})$ (both denote the function that maps each store d to $\{x = d(y)\}$). That is, *w is oblivious to extra reads*. . However we would like to distinguish the two for the first is potentially in a race with a thread writing to `z` whereas the second is not.

We use **partial functions** that record for each **partial store** d the writes produced by executing s on d. On an input store d, the output store $[\![s]\!](d)$ is defined at variable x only if x is written by s. If d does not define a value for a variable read by s, the output will not be defined, thus the function may itself be partial. We also have a simple technique to find out which variables must be read by the function to produce an output. Dually, if $[\![s]\!](d)$ is not defined on a given (partial) store d, then it must mean that s must read some variable that does not have a binding in d.

Let us call such a function for a sequential program s the *partial write function* for s, $pw(s)$. Note that $pw(\texttt{x}=\texttt{z};\texttt{x}=\texttt{y})$ is defined only for those stores d s.t. $\{\texttt{z},\texttt{y}\} \subseteq dom(d)$, whereas $pw(\texttt{x}=\texttt{y})$ is defined for more stores, namely those stores d for which $\{\texttt{y}\} \subseteq dom(d)$. Thus $pw(\texttt{x}=\texttt{z};\texttt{x}=\texttt{y})$ is not the same as $pw(\texttt{x}=\texttt{y})$.

Recall *TStore* is $\{d \mid dom(d) = V\}$. Note that $w(s) = pw(s) \downarrow$ *TStore*, and, by composition, $io(s) = pw(s)^\sharp \downarrow$ *TStore*. Therefore we feel justified in defining (for a partial function f on finite stores) $w(f)$ to be $f \downarrow$ *TStore*, and $io(f)$ to be $f^\sharp \downarrow$ *TStore*.

Sequential functions. This section may be skipped on a first reading. To prepare for LI, it defines $n(f,d)$, the set of variables the step f *must read* in an input store d on which it is not defined. LI will be required to introduce a link labeled with a variable x into a step f only if $x \in n(f, in(f))$, where $in(f)$ is the store obtained by examining incoming links. The definition of $n(f,d)$ takes advantage of the development in denotational semantics of the notion of a sequential (or stable) function [Vui74,Cur93].

Intuitively sequential functions correspond to the execution of single-threaded code. Such code must perform its basic operations (e.g. reads and writes) in sequence, one after the other. (It may not perform operations such as a "parallel or", which reads reads two variables in parallel, without specifying the order.) Therefore such functions f have the property that any store d is either in f's domain, or there is a non-empty set of variables, $n(f,d)$, all variables in which must be read *next* by the function, as we now discuss.

How can we determine whether a particular function f on partial stores corresponds to sequential code? We must formalize the idea "one step at a time, in sequence." [Vui74,Cur93]. The set of stores on which f is *not* defined provides us with the structure we need. If f is not defined on d it must be because f wishes to read a variable which is not defined in d. In fact we can look at the variables $dom(d') \setminus dom(d)$ for some $d' \geq d$ on which f *is* defined as an indication of the set of variables f *may* need to read. (For instance, if f is the step corresponding to the program `r0=x; if (r0==1) r1=y`, then f may need to read `y` in the empty store – whether it actually does or not depends on the value it reads for `x`.) Now let $n(f,d)$ be the intersection of such sets of variables for every $d' \geq d$ on which f is defined. This is the set of variables that f *must necessarily* read. (For instance, for f defined as above, $n(f,\{\}) = \{\texttt{x}\}$. Similarly, $n(f,\{\texttt{x}=1\}) = \{\texttt{y}\}$. We are not interested in $n(f,\{\texttt{x}=0\})$ since f is defined for $\{\texttt{x}=0\}$.) The sequential functions should be those for which this set is non-empty, that is, for every $d \notin dom(f)$ there is a single, determinate way to move forward – read the variables in $n(f,d)$.

DEFINITION 1 (SEQUENTIAL FUNCTION). A partial function f over stores is *sequential* if for every store $d \notin dom(f)$

$$n(f,d) \stackrel{def}{=} \bigcap \{dom(d') \setminus dom(d) \mid d' \geq d, d' \in dom(f)\}$$

is non-empty.

Here are some more examples of the computation of n.

EXAMPLE 4 (Sequential functions) Consider the function f generated by `x=x`.

This function is not defined for $\{\}$. It is defined for $\{\texttt{x}=1\}$, so this is an indication that the function may need to read `x`. It is not hard to see that *any* store on which f is defined must define a value for `x`. Indeed, a direct calculation establishes $n(f,d) = \{\texttt{x}\}$ for any d on which f is not defined.

Next, consider the function f generated by:

```
r=x; r1=(r!=42)?y;x=(r!=42)?r1;
```

$dom(f)$ contains all those stores d such that $\texttt{x} \in dom(d)$, and $\texttt{y} \in dom(d)$ if $d(\texttt{x}) \neq 42$. An easy calculation establishes that for $d \notin dom(f)$, $n(f,d)$ equals $\{\texttt{x}\}$ if $\texttt{x} \notin dom(d)$, and $\{\texttt{y}\}$ if $\texttt{x} \in dom(d), d(\texttt{x}) \neq 42$. □

The last example establishes that $n(f,d)$ is indeed a function of d, and not just f. If one were to think of $n(f,d)$ as providing "type information" for f, then the type of f has to be a *dependent type*: it depends on the argument to f.

An example of a function f that is *not* sequential is one which is defined on an input store d iff *either* $\texttt{x} \in dom(d)$ or $\texttt{y} \in dom(d)$, for two distinct variables `x` and `y`. For such a function $n(f,\{\})$ is empty!

Note that since $n(f,d) = \{\}$ for any total store d (since $dom(d) = V$), all sequential functions are complete.

Step. We now come to the main definition of this section.

DEFINITION 2 (STEP). A *step* is a monotone, sequential, partial function from finite stores to finite stores.

2.3 Modeling concurrent programs

A concurrent program can now be thought of as a *partially ordered multiset (pomset)* of steps. The partial order is called the *happens before* order and indicates those steps that are known to occur before other steps. Formally, an AO process is a initialized pomset of steps, with a possibly empty set of links:

DEFINITION 3 (SEQUENTIAL COMPOSITION). Given two steps f and g, their *sequential composition* $g \circ f$ is the partial function defined only on those d s.t. $d \in dom(f)$ and $f^\sharp(d) \in dom(g)$ and which maps d to $f(d)[g(f^\sharp(d))]$.[5]

While the definition of a step captures only the actual output of the step, the use of a step in a sequential composition permits inputs to traverse untouched to the output of the first step, if they are needed by the second step. However, the output produced by the composite is only the (combination of) output produced by each step – flow through from the input is not counted as output. It is not hard to see that $(f \circ g)^\sharp = f^\sharp \circ g^\sharp$. Examples of sequential composition are provided in Section 3.3.

DEFINITION 4 (LINK). Given a pomset of steps P, a *link* is a quadruple (s,t,x,v) where $s,t \in P$, x is a variable and v is a value.

DEFINITION 5 (INPUT STORE, LINK-COMPLETED STEP). Given a set L of links $(\_,s,x,v)$ entering s, $in(s)$ is the store $\{x = v \mid (\_,s,x,v) \in L\}$. When used as a function, $in(s)$ stands for the function that maps input d to $d[\{x = v \mid (\_,s,x,v) \in L\}]$.

We say that a step s is *complete* if $in(s) \in dom(s)$.

We define $s^\dagger$ (read: link-completed s) as the function $s \circ in(s)$.

DEFINITION 6 (WRITE-BEFORE). Let P be a pomset of steps. For steps $f,g \in P$ and a variable x, define $f\ wb_x\ g$ (read: g can read x from f) if (i) f writes x, i.e. $x \in dom(f(\{\}))$, and (ii) f and g are unordered, or $f\ hb\ g$ and (iii) there is no other step f' between f and g (in the hb-order) s.t. $x \in dom(f'(d))$ for any store d.

DEFINITION 7 (AO PROCESS). An AO process (P,Ls) is a partially ordered multiset of steps P, together with a set Ls of links satisfying:

Link Uniqueness $(s,t,x,v),(s',t,x,v') \in Ls$ implies $s = s'$ and $t = t'$.

Link Well-definedness $(s,t,x,v) \in Ls$ implies s is complete and $s^\dagger(in(s))(x) = v$. (Thus s unconditionally produces v for x, given its input links.)

Link Acyclicity The graph with steps as nodes and edges $s \rightarrow t$ if $(s,t,\_,\_) \in Ls$ is acyclic.

HB Consistency $(s,t,x,v) \in Ls$ implies $s\ wb_X\ t$.

Initialization Condition: If a step in P touches a variable $\mathtt{x} \in V$ then there is a unique step in P that writes into x, does not read from x, and hb any other step in P that touches x.

It is useful to visualize an AO process as a directed graph with nodes labeled with steps and edges representing the hb relation.

DEFINITION 8 (COMPLETED AO PROCESS). An AO process A is said to be a *completed execution* if every step of A is complete.

DEFINITION 9 (SC (EXECUTIONS OF) AO PROCESS). An AO process A is said to be *sequentially consistent* (SC) if its hb order is total. An *SC execution* of an AO process A is any SC AO process A' with the same set of steps and link-set as A.

DEFINITION 10 (WELL-BEHAVED AO PROCESS). An AO process P is *well-behaved* if all its SC executions are race-free.

[5] Note: We have defined $\circ$ to use application order, $f \circ g = \lambda d.(f(g(x)))$, rather than textual order, $f \circ g = \lambda d.(g(f(x)))$.

Process combinators

AO processes are composed using "`;`" (sequential composition) and "`|`" (parallel composition). `;` binds more tightly than `|`.

P ; Q has the steps of P and Q with the hb order of P and Q extended to ensure that every step of P hb every step of Q.

$P \mid Q$ has the steps of P and Q with the hb order of P and Q.

Note that `;` is associative, whereas `|` is commutative and associative (but not idempotent – the resulting pomset has twice as many steps). If we use `skip` to denote the unique process with no steps, then $\mathtt{skip} \mid P = P \mid \mathtt{skip} = P$, and $\mathtt{skip};P = P;\mathtt{skip} = P$.

2.4 Transformations of AO processes

In the RAO model, the following transformation rules can be used to transform an AO process. The transformation is applicable only if the resulting structure is an AO process.

The transformations IM, AU, CO, DL and LI are local, i.e. the applicability of the transformation does not depend on whole program analysis or on the absence or presence of other steps than the ones named in the transformation.

Below, for a process (P,Ls) with steps $p \in P, p'$ when we say *replace p by p′ while preserving all edges and links* we mean that a new process (P',Ls') is created in which P' is the same as P with p replaced by p', every edge $(h,p) \in hb$ is replaced by (h,p'), every edge $(p,h) \in hb$ is replaced by (p,h'), every link $(q,p,x,v) \in Ls$ is replaced by (q,p',x,v), and every link $(p,q,x,v) \in Ls$ is replaced by (p',q,x,v).

2.4.1 Improvement

We say that *a step g improves a step f* if $io(g) = io(f)$, $dom(g) \subseteq dom(f)$, and $f \geq g$. The first condition ensures that the behavior of f and g under sequential (sequentially consistent) execution is identical. The second condition ensures that *extra reads* – reads of variables that do not affect the final result – can be dropped. The third condition ensures that *extra writes* – writes of the form x=x – can be dropped. Let us write $[\![s]\!]$ for the step corresponding to a piece of sequential code s. Then $[\![\mathtt{x} = \mathtt{y}]\!]$ improves $[\![\mathtt{x} = \mathtt{z}; \mathtt{x} = \mathtt{y}]\!]$ and $[\![\mathtt{x} = \mathtt{y}; \mathtt{z} = \mathtt{z}]\!]$.

DEFINITION 11 (IM). Given an AO process (P,Ls), replace $f \in P$ with a step g while preserving all edges and links, if g improves f, and g writes on every variable x for which $(f,\_,x,\_) \in Ls$.

2.4.2 Augmentation

DEFINITION 12 (AU). Add an hb-edge between two steps in P provided that the resulting set is an AO-process.

AU permits the implementation to schedule two otherwise unconstrained steps (belonging to separate threads) in a particular order. As above the application of this transformation does not depend on whole program analysis or on the absence or presence of steps in P other than f and g.

2.4.3 Composition

Consider two steps $f;g$. We would like to replace them with $e = g \circ f$ and move the incoming and outgoing links of f and g to e. That is, we would like to replace $h' = g^\dagger \circ f^\dagger$ by $h = (g \circ f)^\dagger$.

The following conditions are sufficient. If f and g have incoming links for x, those links must arise from the same step (so they read the same value and have the same hb relationship with the link source). This implies $in(f)[in(g)] = in(g)[in(f)]$, or (in terms of functions) $in(f) \circ in(g) = in(g) \circ in(f)$. Further, f should pass through, without modification, any variable for which there is a link into g. Symmetrically, if f has an outgoing link for x, then g should pass through the value produced by f on x without modification. This motivates the following definition.

DEFINITION 13 (CO). Let (P,Ls) be an AO process. Let the immediate hb successor of f in P be g (and only g), and the immediate hb predecessor of g be f (and only f). Let $h = g \circ f \circ in(g) \circ in(f)$ and $h' = g \circ in(g) \circ f \circ in(f)$. Let f and g satisfy the property that (i) $(s,f,x,v),(s',g,x,v') \in Ls$ implies $s = s'$ (and therefore $v = v'$), (ii) $h = h'$, (iii) for every x s.t. $(f,\_,x,v) \in Ls$, $h(in(f)[in(g)])(x) = f(in(f)[in(g)])(x)$.

Replace f and g by $e = g \circ f$, replacing each link/edge entering/exiting f or g by the same link/edge entering/exiting e.

CO permits the implementation to schedule two successive steps in the hb-order together, treating them as part of the same sequential step. In the new process $e^\dagger$ is the same function as $g^\dagger \circ f^\dagger$ in the old process. Further, the conditions are always satisfied if g has no incoming links and f has no outgoing links.

Note that this is a local transformation. Its applicability does not depend on whole program analysis or on the absence or presence of other steps in P than the ones named in the transformation.

2.4.4 Decomposition

DEFINITION 14 (DL). Let (P,Ls) be an AO process, $f \in P$ s.t. $f = h \circ g$, and for every incoming (outgoing) x-link for f it is the case that precisely one of f or g reads (writes) x. (Call that step i_x.)

Replace f with $g;h$. Every edge $(e,f) \in hb$ is replaced by (e,g) and every edge $(f,e) \in hb$ by (h,e). Every link (e,f,x,v) and (f,e,x,v) in Ls is replaced by (e,i_x,x,v) and (i_x,e,x,v) respectively.

Intuitively, the implementation decides to break up a single step f into two steps g and h since the behavior of a thread executing f is indistinguishable (in any race-free context) from the behavior of the thread executing first g and then h.

DR adds to DL the condition that for any variable x and input store d, x is in the domain of at most one of the stores $g^\dagger(d)$ and $(g^\dagger \circ h^\dagger)(d)$. DW adds to DL the condition that for any variable x and input store d, x is in at most one of $n(g^\dagger,d)$ and $n(g^\dagger \circ h^\dagger,d)$. DO adds both these conditions to DL. We let DX stand for any of these four decomposition rules.

Like CO, DX is local. In combination with CO, DX may change the hb order of the original program. For instance consider the program fragment `x=1;y=2` (continuing with our use of an informal notation). Using CO this may be converted to `x,y=1,2` and then using DX to `y=2;x=1`. Thus the original hb order is inverted. Some synchronization constructs (e.g. volatiles) are designed to ensure that such reordering cannot occur (see Section 5.3); hence their semantics places restrictions on the application of DX.

DX is also useful in conjunction with LI: sometimes it is possible to break a function f which performs some reads into f_0 and f_1 in such a way that f_0 does not perform any reads. Now f_0 can be used as a source for a link.

2.4.5 Link

LI is an "inter thread" version of CO.

DEFINITION 15 (LI). Let $A = (P,Ls)$ be a process. Let $s,t \in P$ and x be a variable s.t. (i) $s\ wb_X\ t$ in P, (ii) s is completed, (iii) $s(in(s))(x) = v$, and (iv) $x \in n(t,in(t))$.

Transform A to $(P,Ls \cup \{(s,t,x,v)\})$.

LI is the only way of introducing links. Note that Condition (iv) above is strong enough to ensure Link Acyclicity for any AO process.

2.4.6 Propagation

By a constraint q on stores we mean a (possibly infinite) set of stores. A store d satisfies q if $d \in q$. Two functions f_0 and f_1 on stores are q-equivalent if for $Q = q \cap dom(f_0)$, we have $Q = q \cap dom(f_1)$, and $f_0 \downarrow Q = f_1 \downarrow Q$.

DEFINITION 16 (PR). Let $A = (P,Ls)$ be a process. Let $f \in P$ and f' be a step that is q-equivalent to f, where in all SC-executions of P, q is true at (before) f.

Replace f by f', preserving all edges and links.

(Recall the notion of SC-execution of P is specified in Definition 9.) PR permits an implementation to perform any global optimization based on data-flow analyses as long as the analyses consider only SC executions. Since this transformation effects a global analysis, it is sensitive to the presence steps in P other than s. One of its uses is to replace conditional execution with unconditional execution.

We shall see below that typically an application of CO enables applications of DL. Applications of PR and AU enable applications of CO. Applications of DL enable applications of AU, etc.

2.4.7 RAO process

DEFINITION 17 (RAO PROCESS). An RAO *process* is a set of AO processes closed under CO, DL, IM, LI, PR and AU. For any AO process P, the smallest (qua set) RAO process containing P is denoted by RAO(P).

DEFINITION 18 (EXECUTION). An *execution* of an AO process P is any complete process $P' \in$ RAO(P).

2.5 Main theorem

Let P,Q be AO processes. Say that $P \xrightarrow{X} Q$ if Q is obtained from P by the application of a transform X in the set of RAO transformations. The *SC i/o functions* of P, $sc(P)$ is the set of functions $io(s_0 \circ \ldots \circ s_{n-1})$ where $\{s_0,\ldots,s_{n-1}\}$ is a totally ordered extension of P (with steps enumerated in hb -order).

Let $clo(P)$ represent the set of AO processes obtained from P by zero or more applications of the given transformation. We take the *observations* of a process P to be the set of *i/o functions* of P, $io(P)$ defined as the set $\{f \mid f \in sc(Q), Q \in clo(P), Q \text{ complete}\}$. We say that $O \in io(P)$ has *a proof of size n* if there is an $\xrightarrow{X}$ sequence of length n from P to a completed process Q such that $O \in sc(Q)$.

The proofs of the following lemmas and theorems is in Appendix A.

LEMMA 19 (GOOD BEHAVIOR IS $\xrightarrow{X}$-INVARIANT.). *For all* AO *processes P,Q if P is well-behaved and $P \xrightarrow{X} Q$ then Q is well-behaved.*

LEMMA 20 (PRESERVATION OF SC BEHAVIOR). *Let P,Q be* AO *processes such that $P \xrightarrow{X} Q$, and P is well-behaved. Then* $\mathrm{sc}(Q) \subseteq \mathrm{sc}(P)$.

LEMMA 21 (MAIN LEMMA). *For all* AO *processes P,Q if P is well-behaved and $P \xrightarrow{X} Q$ then* $\mathrm{io}(Q) \subseteq \mathrm{sc}(P)$.

LEMMA 22. *For all* AO *processes P,Q if P is well-behaved and $P \xrightarrow{X} Q$ then:*

Good behavior is $\xrightarrow{X}$-invariant. *Q is well-behaved.*
SC behavior is $\xrightarrow{X}$-invariant. $\mathrm{sc}(Q) \subseteq \mathrm{sc}(P)$.
IO behavior is $\xrightarrow{X}$-invariant. $\mathrm{io}(Q) \subseteq \mathrm{sc}(P)$.

THEOREM 23 (FUNDAMENTAL PROPERTY). *Let P be a well-behaved* AO *process. Then* $\mathrm{io}(P) \subseteq \mathrm{sc}(P)$.

3. The RAO calculus

We now introduce a syntax for steps and show how to associate with a syntactic step s its denotation, $[\![s]\!]$. Further we provide a rule (Proposition 24) that shows how to calculate the sequential composition $s_1;s_2$ of two syntactic steps, and eliminate the ";". The formal treatment of the causality test cases is based on this proposition.

The syntax is intended to be illustrative; it may be extended routinely with concepts such as function definitions. It is not intended to be a full-fledged programming language. Rather it is intended to have a core set of constructs into which a concrete programming language can be translated so that its memory model can be defined.

(Variables)	x	::=	$\mathtt{x} \mid \ldots$
(Condition)	c	::=	$\mathtt{true} \mid \mathtt{false} \mid e{=}{=}e \mid c\&\&c \mid !c$
(Expression)	e	::=	$\mathtt{k} \mid x \mid c\,?\,e\,{:}\,e \mid c\,?\,e \mid (e)$
(Step)	s	::=	$\bar{x}{=}\bar{e}$

The language is simple. It permits one-sided and two-sided conditional definition of terms. We will write `skip` for the step $\varepsilon = \varepsilon$.

3.1 `defined` and `value`.

To define the function that is denoted by a step, we need two central notions, the *definedness* of a step in a given store, and the *value* of an expression in a given store. We shall define these using structural induction. The definitions are standard. For convenience, we use the language of definite clauses (Prolog) to state these definitions.

```
defined((x1,...,xn=e1,... en),d) if
 defined(e1,d),..., defined(en,d).
defined(k,d). defined(x,d) if x in dom(d).
defined(c?e1:e2,d) if
 defined(c, d), (value(c,true,d), defined(e1,d) ;
                 value(c,false,d), defined(e2,d)).
defined(c?e,d) if defined(c,d),
                  (value(c,true,d), defined(e,d);
                   value(c,false,d)).
defined(t1==t2,d) if defined(t1,d), defined(t2,d)
defined(c1&&c2,d) if defined(c1,d), defined(c2,d)
defined(!c, d) if defined(c,d).

value(k,k,d).
value(x,v,d) if d(x)=v.
value(c?e1:e2,v,d)
               if (value(c,true,d),value(e1,v,d);
                   value(c,false,d),value(e2,v,d)).
value(c?e1,v,d) if value(c,true,d), value(e1,v,d).
```

Intuitively, a step is defined in a store d if the store is defined for all variables that will be read by the step. Some care must be taken to ensure that this definition is precise – the variables to be read must not be under- or over-approximated. For instance, if the conditional c of an expression $c?e_1 : e_2$ evaluates to `true` in the given input store, then e_2 will not be evaluated. Thus the variables to be read in the store may depend on the values of other variables in the store – as indicated by the dependence of `defined` on `value`.

Note that `value` is partially defined, that is, $\mathtt{value}(e,v,d)$ may fail to be provable for a given e and d. This is the case if $e = c?e_1$ for instance, and the store evaluates c to `false`.

3.2 Denotation of a step.

We can now define the function $[\![\_]\!]$, which maps a syntactic step $s \equiv \bar{x} = \bar{e}$ to its denotation $[\![s]\!]$, thus: its domain consists of those input stores d for which each $e \in \bar{e}$ is defined. It maps such a d to $d' = \{x = v \mid \mathtt{value}(e,v,d), i < n, x = e \in s\}$. That is, the store d' contains all the writes actually produced by the step – possibly none – given the input store d.

EXAMPLE 5 $[\![\mathtt{x} = (\mathtt{false?0})]\!]$ is the unique function that is defined on every input (i.e. it does not perform a read) and maps it to $\{\}$. In no store can the assignment to `x` happen since its precondition, `false`, can never be satisfied. Formally, for any `v` and `d`, `value(false?0,v,d)` is undefined (fails), .

$[\![\mathtt{x} = 1]\!]$ is the function that is defined on every input and maps it to $\{\mathtt{x} = 1\}$.

Let us now consider some examples with conditionals. On any input store d, $f = [\![\mathtt{x} = (\mathtt{y} == 1?1)]\!]$ is a function that must definitely read `y`, hence d must define a value for `y`. The function produces a write on `x`, `x=1`, iff $d(\mathtt{y}) = 1$. Formally, $d \in dom(f)$ iff $\mathtt{y} \in dom(d)$. f maps such a d to $\{\mathtt{x} = 1\}$ if $d(\mathtt{y}) = 1$ and to $\{\}$ otherwise.

$[\![x = (y == 1?0 : (y == 0?1))]\!]$ is defined on input stores d iff $\mathtt{y} \in dom(d)$. Such a d is mapped to $\{\mathtt{x} = 0\}$ if $d(\mathtt{y}) = 1$, to $\{\mathtt{x} = 1\}$ if $d(\mathtt{y}) = 0$ and to $\{\}$ otherwise.

$f = [\![x, r = (x! = 42?42), 42]\!]$ is defined on input stores d iff $\mathtt{x} \in dom(d)$. Such a d is mapped to $\{\mathtt{x} = 42, \mathtt{r} = 42\}$ if $d(\mathtt{x})! = 42$ and to $\{r = 42\}$ if $d(\mathtt{x}) = 42$. Note that for all $d \in dom(f)$ we have $\{\mathtt{x} = 42\} \leq f^\sharp(d)$ – in some cases because of the write in $f(d)$ and in some cases because of the flow-through from the input. Our treatment of steps as partial functions enables us to model this distinction.

□

Definition of n. We can now given an explicit definition of n on syntactic steps in a way that is consistent with their denotation. That is, for any syntactic step p, $n(p,d) = n([\![p]\!],d)$, for all stores d.

$$
\begin{array}{rcl}
n(\mathtt{x} = \mathtt{k}, d) & = & \emptyset \\
n(\mathtt{x} = \mathtt{y}, d) & = & \{\mathtt{y}\} \\
n(\mathtt{x} = (\mathtt{e}), d) & = & n(\mathtt{x} = \mathtt{e}, d) \\
n(\mathtt{x} = \mathtt{c}\,?\,\mathtt{e}_0 : \mathtt{e}_1, d) & = & \left\{ \begin{array}{l} n(\mathtt{x} = \mathtt{e}_0, d) \\ \qquad \text{if } \mathtt{value}(\mathtt{c}, \mathtt{true}, d) \\ n(\mathtt{x} = \mathtt{e}_1, d) \\ \qquad \text{if } \mathtt{value}(\mathtt{c}, \mathtt{false}, d) \\ \mathbf{var}(c) \setminus dom(d), \text{ o.w.} \end{array} \right. \\
n(\mathtt{x} = \mathtt{c}\,?\,\mathtt{e}_0, d) & = & n(\mathtt{x} = \mathtt{c}\,?\,\mathtt{e}_0 : \mathtt{0}, d) \\
n(\mathtt{x}_0, \ldots, \mathtt{x}_{n-1} = \mathtt{e}_0, \ldots, \mathtt{e}_{n-1}, d) & = & n(\mathtt{x}_0 = \mathtt{e}_0, d) \cup \ldots \cup n(\mathtt{x}_{n-1} = \mathtt{e}_{n-1}, d)
\end{array}
$$

3.3 Calculating sequential composition of syntactic steps.

We now consider examples of sequential compositions of steps.

EXAMPLE 6 Consider $f = [\![\mathtt{x} = 1]\!]; [\![\mathtt{y} = 1]\!]$. It is not difficult to see that $f = [\![\mathtt{x}, \mathtt{y} = 1, 1]\!]$. Formally, one uses the definition of denotation of a step given above, and the definition of composition of steps (Definition 3) to establish this.

Let us consider a step that reads a variable after conditionally writing into it: $f = [\![\mathtt{x} = (\mathtt{x} == 1?0 : 1)]\!]; [\![\mathtt{y} = \mathtt{x}]\!]$. Clearly this should be the same function as $[\![\mathtt{x}, \mathtt{y} = (\mathtt{x} == 1?0 : 1), (\mathtt{x} == 1?0 : 1)]\!]$. Again, this can be established formally.

In general, $[\![\mathtt{x} = c?\mathtt{z}]\!]; [\![\mathtt{y} = \mathtt{x}]\!]$ is the same as $[\![\mathtt{x}, \mathtt{y} = c?\mathtt{z}, c?\mathtt{z} : \mathtt{x}]\!]$. One reasons as follows. Note that both expressions are defined for stores d iff c is defined in d and $\mathtt{x}, \mathtt{z} \in dom(d)$. Now there are two cases. If $\mathtt{value}(\mathtt{c}, \mathtt{true}, \mathtt{d})$, then $\mathtt{x} = c?\mathtt{z}$ produces the output $\{\mathtt{x} = \mathtt{z}\}$ and this overrides the value of `x` in d. Hence the value of `y` in the resultant store will be `z`. If $\mathtt{value}(\mathtt{c}, \mathtt{false}, \mathtt{d})$, $\mathtt{x} = c?\mathtt{z}$ produces the output $\{\}$, and the value of `x` read by `y` will be $d(\mathtt{x})$.

Specifically $[\![\mathtt{x} = (\mathtt{x}! = \mathtt{k}?\mathtt{k}); \mathtt{y} = \mathtt{x}]\!]$ is the same as $[\![\mathtt{x}, \mathtt{y} = (\mathtt{x}! = \mathtt{k}?\mathtt{k}), \mathtt{x}! = \mathtt{k}?\mathtt{k} : \mathtt{x}]\!]$, and this is the same as $[\![\mathtt{x}, \mathtt{y} = (\mathtt{x}! = \mathtt{k}?\mathtt{k}), \mathtt{k}]\!]$. □

We can formalize a rule for calculation of sequential composition of steps as follows. Let the term $s\{t\}$ be defined to be the term s except that every single-armed conditional $c\,?\,e$ in s is mapped to $c\,?\,e : t$. Formally:

$$
\begin{array}{rcl}
\mathtt{k}\{t\} & = & \mathtt{k} \\
\mathtt{x}\{t\} & = & \mathtt{x} \\
(e)\{t\} & = & e\{t\} \\
c\,?\,e_0 : e_1\{t\} & = & c\{t\}\,?\,e_0\{t\} : e_1\{t\} \\
c\,?\,e\{t\} & = & c\{t\}\,?\,e\{t\} : t \\
t_0{=}{=}t_1\{t\} & = & t_0\{t\}{=}{=}t_1\{t\} \\
!c\{t\} & = & !(c\{t\}) \\
(c_0 \&\& c_1)\{t\} & = & c_0\{t\} \&\& c_1\{t\}
\end{array}
$$

The following proposition shows how to simply a sequential composition of steps $s_1;s_2$ by eliminating ";" and justifies our use of the term "memory model calculus."

PROPOSITION 24 (ELIMINATING SEQUENTIAL COMPOSITION). *$[\![\bar{x},\bar{y},\bar{z}=\bar{s},(\bar{u}\theta)\{\bar{t}\},\bar{v}\theta]\!]$ (where θ is the substitution $[\bar{s}\{\bar{x}\}/\bar{x},\bar{t}\{\bar{y}\}/\bar{y}]$) is an improvement on $[\![\bar{x},\bar{y}=\bar{s},\bar{t}]\!] \circ [\![\bar{y},\bar{z}=\bar{u},\bar{v}]\!]$, given that $\bar{\mathtt{x}},\bar{\mathtt{y}}$ and $\bar{\mathtt{z}}$ are pairwise disjoint.*

Above, we take $\bar{u}\{\bar{t}\}$ to be shorthand for $u_0\{t_0\},\ldots,u_{n-1}\{t_{n-1}\}$, if $u=u_0,\ldots,u_{n-1}$ and $t=t_0,\ldots,t_{n-1}$.

The semantic step $[\![\bar{x},\bar{y},\bar{z}=\bar{s},(\bar{u}\theta)\{\bar{t}\},\bar{v}\theta]\!]$ is an improvement only because it may perform fewer reads. For instance, the syntactic step s given by $\mathtt{x}=\mathtt{y};\mathtt{x}=2$ can be simplified, by applying the proposition, to s' given by $\mathtt{x}=2$.[6] It is possible to formulate a precise rule for eliminating sequential composition at the cost of introducing new local variables that are assigned terms that perform the missing extra reads. There is no practical reason for doing so since all the models we consider admit the transformation IM which permits a step to be improved.

EXAMPLE 7 The answers in Example 6 can be calculated using this proposition. □

3.4 Calculating DX

In terms of concrete syntax, a step $\bar{x},\bar{y}=\bar{u},\bar{v}$ can be decomposed via DL into $\bar{x}=\bar{u}$ and $\bar{y}=\bar{v}$. DR requires that $\bar{x}$ and $\bar{y}$ are disjoint. DW requires that the variables in u and v are disjoint. DO requires both conditions.

3.4.1 Calculating IM

A step $x,\bar{y}=x,\bar{u}$ may be replaced by $\bar{y}=\bar{u}$. That is, extra writes can be dropped. The rule for sequential composition already already drops extra reads.

3.5 Calculating LI

We now consider examples of merging.

EXAMPLE 8 Below we assume that f,g are two steps in an AO process (P,Ls) and the only link entering g is from f and labeled with x.
Let f be `x=1` and g `r=x`. Then $g^\dagger$ is `r=1`. f can be used to answer the read on `x` in g, but f's outputs are not propagated.
Let f be `x,y=1,1` and g `r=x`. Then $g^\dagger$ is `r=1`. Irrelevant information in f is ignored.
Let f be `x=42` and g `r,x=x,(x!=42)?42`. Then $g^\dagger$ is `r=42`. Information in f may force a write of g to be dropped.
Let f be `x=(x!=42)?42` and g `r=x`. Then `r=42` is an improvement of $g^\dagger$ (since it does not read `x`. A conditional write in f may result in an unconditional write by g.

□

[6] In detail, the result is `x=2[y{x}/x] {y}`. Now note that $\mathtt{y}\{\mathtt{x}\}=\mathtt{y}$, and $2[\mathtt{y}/\mathtt{x}]=2$. Finally $2\{\mathtt{y}\}=2$. s' is an improvement of s because it does not read `y`.

PROPOSITION 25 (ELIMINATING MERGE). *Let f be the step $[\![\bar{x},\bar{y}=\bar{s},\bar{t}]\!]$, and g the step $[\![\bar{y},\bar{z}=\bar{u},\bar{v}]\!]$ where $\bar{\mathtt{x}},\bar{\mathtt{y}}$ and $\bar{\mathtt{z}}$ are pairwise disjoint, and for each variable in $\bar{y}$ there is a link from f to g (and these are the only links). Then $[\![\bar{y},\bar{z}=(\bar{u}\theta)\{\bar{t}\},\bar{v}\theta]\!]$ improves on $g^\dagger$, where θ is the substitution $[\bar{s}\{\bar{x}\}/\bar{x},\bar{t}\{\bar{y}\}/\bar{y}]$.*

Note that a step $[\![\bar{x},\bar{y}=\bar{s},\bar{t}]\!]$ can be the source of a link only if for each term $s\in\bar{s},\bar{t}$, `defined(s,{})`. This is the syntactic rendition of the semantic condition that for f to be the source of a link it must be complete.

4. Examples

We consider some examples. Note: In analyzing the test cases below we shall usually omit the initial step in the AO process. Further, we shall not be combining (through CO) two steps both of which have incoming links. In such cases it is possible to replace t with $t^\dagger$ whenever a new link (s,t,x,v) is added to the link-set.

4.1 Single-thread reordering

EXAMPLE 9 (TC 7) We illustrate the use of CO, DE and AU to obtain single-thread reordering. Consider the program:

```
x,y,z=0,0,0;(r1=z; r2=x; y=r2 | r3=y; z=r3; x=1)
```

Is behavior `r1=r2=r3=1` exhibited? Single-thread optimization could permit `r1=z` to be moved to the end of the thread, and `x=1` to the beginning of the thread. The result would then follow by an SC execution.

Formally this can be analyzed as follows. We show a chain of AO processes each obtained from the previous by applying the noted transformation. The last process exhibits the desired behavior.

Consider the steps `r1=z; r2=x; y=r2`. These may be collapsed into a single step using CO to yield `r1,r2,y=z,x,x`. But this step can be decomposed into `r1=z | r2,y=x,x` – this is the code motion discussed above. Similarly `r3=y;z=r3;x=1` yields through CO and DE `r3,z=y,y | x=1`. Now we can interleave the steps in the appropriate order using AU to accomplish the desired result.

In detail, the derivation is as follows. In all examples we use Proposition 24 to eliminate sequential composition when applying CO. In each step we specify only the links added at that step. By convention the links associated with a step are the union of all the links associated with previous steps, together with the links added at that step.

```
x,y,z=0,0,0; (r1=z; s1: r2=x; y=r2
                | r3=y; z=r3; s5: x=1)
x,y,z=0,0,0;(r1=z; s1: r2=1; y=r2
                | r3=y;z=r3;s5: x=1)      (LI, (s5,s1))
x,y,z=0,0,0;(r1=z; s1: r2,y=1,1
                | r3=y;z=r3;s5: x=1)      (CO)
x,y,z=0,0,0;(r1=z; s1: r2,y=1,1
                | s2: r3=1;z=r3;s5: x=1) (LI, (s1,s2))
x,y,z=0,0,0;(r1=z; s1: r2,y=1,1
                | s2: r3,z=1,1;s5: x=1)  (LI, (s1,s2))
x,y,z=0,0,0;(s0: r1=1; s1: r2,y=1,1
                | s2: r3,z=1,1; s5: x=1) (LI, (s2,s0))
```

Each of these processes is in $\mathsf{RAO}(P_0)$. The last is a completed execution, and establishes the desired result. □

EXAMPLE 10 (TC 2) See also Fig 5 in [MPA05]. This example illustrates that CO, DE and AU can simulate the effect of redundant read elimination. Consider the program:

```
x,y=0,0; (r1=x;r2=x; y=(r1==r2)?1 | r3=y;x=r3)
```

This should exhibit `r1==r2==r3==1` since redundant read elimination could result in simplifying `r1==r2` to `true`. Subsequently `y=1` could be moved early.

This reasoning is readily formalized as follows. In each step we specify only the links added at that step. By convention the links associated with a step are the union of all the links associated with previous steps, together with the links added at that step.

```
x,y=0,0; (r1,r2=x,x;y=(r1==r2)?1 | r3=y;x=r3)   (CO)
x,y=0,0; (r1,r2,y=x,x,1 | r3=y;x=r3)            (CO)
x,y=0,0; (s0: r1,r2,y=x,x,1 | s1: r3=1;x=r3)
          (LI,(s0,s1))
x,y=0,0; (r1,r2,y=x,x,1 | r3,x=1,1)             (CO)
```

□

This example shows that the RAO model permits two reads to be answered by the same write **without determining what that write is**. This is just a consequence of CO – by composing all the steps of Thread 1, we ensure that the reads into `r1` and `r2` will be answered from the input store (for the composite step). Hence they must have the same value. Thus the steps of the first thread are equivalent (as functions) to the single step `r1,r2,y=x,x,1`.

EXAMPLE 11 (TC 3) This example illustrates that the application of CO, DE and AU is not affected by the presence of additional threads. Consider the program:

```
x,y=0,0;
(r1=x; r2=x; y=(r1 == r2)?1 | r3=y; x=r3; | x=2
```

The behavior `r1 == r2 == r3 == 1` can be exhibited, using the same reasoning as in Test 10. The additional thread does not interfere with the application of CO and LI. □

EXAMPLE 12 (TC 17) Consider the AO process:

```
x,y=0,0; (r3=x; x=(r3!=42)?42; r1=x;y=r1 | r2=y;x=r2)
```

It should be able to exhibit `r1==r2==r3==42` since `r3=x; x=(r3 != 42)?42; r1=x` and `r3=x; x=(r3 != 42)?42; r1=42` have identical i/o functions. But the second program can permit the propagation of `r1=42` to the beginning of the program, resulting in the desired behavior. The RAO analysis mirrors this reasoning:

```
r3=x; x=(r3!=42)?42; r1=x;y=r1 | r2=y;x=r2
r3,x=x,(x!=42)?42; r1=x;y=r1 | r2=y;x=r2          (CO)
r3,x,r1=x,(x!=42)?42,42; y=r1 | r2=y;x=r2         (CO#)
r3,x,r1,y=x,(x!=42)?42,42,42 | r2=y;x=r2          (CO)
r3,x=x,(x!=42)?42; s0: r1,y=42,42 | s1: r2=y;x=r2 (DL)
r3,x=x,(x!=42)?42; s0: r1,y=42,42 | s1: r2=42;x=r2
    (LI, s0->s1)
r3,x=x,(x!=42)?42; so: r1,y=42,42 | s1: r2,x=42,42(CO)
 s3: r3=42; s0: r1,y=42,42 | s1: r2,x=42,42
    (LI, s1->s3)
```

(#) In the above example, `r3,x,r1=x,x!=42?42,x!=42?42:x` and `r3,x,r1=x,x!=42?42,42` – denote the same step. In the last line the write to `x` will never be performed by the first step, and hence the write is dropped. □

4.1.1 Inter-thread reasoning – the use of PR

We now consider some examples that illustrate the use of PR.

EXAMPLE 13 (TC 1) This example shows inter-thread reasoning – the use of CO,DE,AU,PR. Consider the RAO process generated from P_0: `x,y=0,0; (r1=x;y=(r1>=0)?1 | r2=y; x=r2)`

Arguably, RAO(P_0) should be able to exhibit `r1==r2==1`. The compiler may determine that `x` and `y` are always non-negative, and hence simplify `r1 >=0` to `true`. This allows `y=1` to be moved early. We can formalize this in RAO thus:

```
r1,y=x,r1>=0?1 | r2=y; x=r2       (CO)
r1,y=x,1 | r2=y; x=r2             (PR#)
r1=x; s1: y=1 | s2: r2=y; x=r2    (DL)
r1=x; s1: y=1 | s2: r2=1; x=r2    (LI, s1->s2)
s0: r1=x; s1: y=1 | s2: r2,x=1,1 (CO)
s0: r1=1; s1: y=1 | s2: r2,x=1,1 (LI, s2->s0)
```

(PR#) Replace `r1,y=x,(x>=0?1)` with the `x>=0`-equivalent step `r1,y=x,1`.

□

As the example above illustrates, RAO permits sophisticated patterns of interaction between PR, CO, SE, AU.

EXAMPLE 14 (TC 18) See also [MPA05, Fig 12]. The program:

```
x,y=0,0; (r3=x; x=(r3==0)?1;r1=x;y=r1 | r2=y;x=r2)
```

should permit the behavior `r1==r2==r3==1`. A compiler may determine through whole program analysis that the only possible values for `x` are `0` and `1`. Hence if `r3 !=0` it must be the case that `r3==1`. Hence transforming `r1=x` into `r1=1` is legal from the viewpoint of a single thread. But this write can be propagated earlier and SC execution will yield the desired result. The RAO analysis permits this, following the reasoning above.

```
r3,x=x,(x==0)?1;r1=x;y=r1|r2=y;x=r2           (CO)
r3,x,r1=x,(x==0)?1,(x==0)?1:x;y=r1|r2=y;x=r2 (CO)
r3,x,r1=x,(x==0)?1,1; y=r1|r2=y;x=r2
    (PR;x in {0,1}
r3,x,r1,y=x,(x==0)?1,1,1|r2=y;x=r2            (CO)
r3,x=x,(x==0)?1;s0: r1,y=1,1|s1: r2=y;x=r2    (DL)
r3,x=x,(x==0)?1;s0: r1,y=1,1|s1: r2=1;x=r2
    (LI, s0->s1)
s2: r3,x=x,(x==0)?1;s0: r1,y=1,1|s1: r2,x=1,1(CO)
s2: r3=1;s0: r1,y=1,1|s1: r2,x=1,1
    (LI, s1->s2)
```

□

EXAMPLE 15 (Fig 11 of [MPA05]) This test case is *not* permitted by the Java Memory Model described in [MPA05], but is permitted by RAO. Consider the program:

```
x,y=0,0; (r3=x; x=(r3==0)?1 | r1=x;y=r1 | r2=y;x=r2)
```

Test Case 18 can be obtained from this program by inlining Thread 2 after Thread 1.

```
x,y=0,0; (r3=x; x=(r3==0)?1 | r1=x;y=r1 | r2=y;x=r2)
x,y=0,0; (r3=x; x=(r3==0)?1 ; r1=x;y=r1 | r2=y;x=r2) (AU)
```

The rest of the derivation follows Case 18. □

4.2 Cross-coupling behaviors

We now consider examples that illustrate *cross-over*.

DEFINITION 26 (CROSS-OVER). Let A be an AO process. A *cross-over* is a set of steps in A that forms a loop in the graph whose nodes are steps and whose edges are links (directed from source to target) or hb-edges.

Naturally, the presence of races, and the use of LI, is critical in establishing a cross-over. The other transformations (CO, DX, IM, AUand PR) are compatible with a totally-ordered notion of memory — memory is a global set of locations from which every read fetches the current value and every write modifies the current value. In our model, such a totally-ordered notion of memory is modelled by the extra condition that n particular, these other transformations are closed on the subset of AO processes satisfying the condition that the linkset is a subset of the happens-before relation. LI does

not preserve this additional condition, whereas the other transformations (CO, DX, IM, AUand PR) do.

EXAMPLE 16 (TC 16) See also Fig 1 in [MPA05]. The program:

```
x,y=0,0; (r1=x; x=1 | r2=x; x=2)
```

should be able to exhibit the behavior `r1==2; r2==1`. RAO permits it thus:

```
x,y=0,0; (s0: r1=x; s1: x=1 | s2: r2=x; s3: x=2)
x,y=0,0; (s0: r1=x; s1: x=1 | s2: r2=1; s3: x=2)
     (LI, s1->s2)
x,y=0,0; (s0: r1=2; s1: x=1 | s2: r2=1; s3: x=2)
     (LI, s3->s0)
```

The final process illustrates the crossover $\{s0, s1, s2, s3\}$. □

We now consider an example that shows the interleaving of LI and PR is critical.

EXAMPLE 17 (Fig 11a) This is a variation on Test Case 15.

```
x,y,z=0,0,0;
(r1=x; x=(r1==0)?1; a=r1; r11=x1; x1=(r11==0)?1
 | r2=x; y=r2; r21=x1; y1=r21
 | r3=y; x=r3; r31=y1; x1=r31
 | r4=x; r5=y; x1=(r4+r5<2)?25)
```

The behavior in question is: `r1==r2==r3==r11==r21==r31==1`.

The code in TC 15 is considered to be part of "Phase 1." A copy of the code is made (on a parallel set of variables) to get a "Phase 2." The results of the first phase are detected and used to determine whether a particular write is allowed in Phase 2 (the code for Thread 4).

If the first phase yields `r1=r2=r3=1`, then there is enough information to rule out the write of `25` into `x1`. Now it can be established that the only permissible values for `x1` are `0` and `1` and the same reasoning used to establish `r1=r2=r3=1` (namely a use of CO and PR) can be used to establish that `r11=r21=r31=1`, thus getting the desired answer. □

4.3 No Thin Air Reads behaviors

The following examples involving no thin air reads discuss alternative definitions of the decomposition rule and their consequences. This analysis supports the claim that RAO provides a flexible framework for a programming language designer.

EXAMPLE 18 (TC 4) See also Fig 2 in [MPA05]. Consider the AO process:

```
x,y=0,0; (r1=x;y=r1 | r2=y;x=r2)
```

This process should not exhibit `r1==r2==1` even though there is a race. The value `1` cannot be read from thin air.

LI, PR and AU cannot produce the desired result, as can be established by systematically applying them.

Now let us consider various decomposition rules. DO (and hence DL) can establish `r1==r2==1` by:

```
x=0;y=0;(r1=x;y=r1 | r2=y;x=r2)
x=0;y=0;(r1=x;y=1;y=r1 | r2=y;x=r2)  (DW)
x=0;y=0;(y=1;r1=x;y=r1 | r2=y;x=r2)  (DO)
x=0;y=0;y=1;r2=y;x=r2;r1=x;y=r1      (AU*)
y=1;r2=1;r1=1;x=1                    (DO*)
```

However, DR and DO cannot; there is no way of creating the phantom write. □

EXAMPLE 19 (TC 5) Consider the program:

```
x,y,z=0,0,0;
(r1=x; y=r1 | r2=y; x=r2 | z=1 | r3=z; x=r3)
```

The behavior `r1==r2==1, r3==0` should be forbidden.

RAO Analysis: As in Test Case 18. The only use of LI will replace `r3=z` with `r3=1` – and this will not give the desired result. An exhaustive case analysis shows that none of the other transformations can produce the desired behavior. □

EXAMPLE 20 (TC 10) Consider the AO program *P*:

```
x=0;y=0;z=0;
( r1=x;y=(r1==1)?1 | r2=y; x=(r2==1)?1
 | z=1 | r3=z;x=(r3==1)?1 )
```

The behavior `r1==r2==1, r3==0` should not be possible.

This is indeed the case. PR cannot be used to discharge any of the conditionals. CO/DE cannot be used to perform any of the steps of a thread in parallel since there is a read/write dependency. AU can be used to totally order these steps (as would be done in an *sc* execution). But no *sc* execution will give the desired result. LI can be used to replace `r3=z` with `r3=1`, but this will not give the desired result.

□

EXAMPLE 21 (TC 13) See also [MPA05, Fig 4]. Consider the program:

```
x=0;y=0; (r1=x; y=(r1==1)?1 | r2=y;x=(r2==1)?1)
```

`r1==r2==1` should not be observed since no writes occur to `x` and `y` in any SC execution, and the program has no races. RAO disallows this, with the reasoning closely following TestCase 20.

□

EXAMPLE 22 (Fig 10, [MPA05]) Consider the program:

```
x=0;y=0;z=0;
(z=1 | r1=z;x=(r1==0)?1 | r2=x;y=r2 | r3=y;x=r3)
```

It should not be possible to observe `r1==r2==r3==1`, since in any "execution" which could exhibit this behavior only Threads 3 and 4 write to `x` and `y`, and hence they cannot manufacture the value `1` out of thin air.

The RAO model validates this reasoning. It is not possible to use PR to reduce `x=(r1==0)?1` to `x=1` (except by using AU to place `z=1` after the conditional assignment to `x` – but in that case `r1=z` *hb* `z=1` hence `r1` can never see the value `1`). Without that, the only way `r2` can be `1` is for `r1=0` to have been executed before it, but then `r1 !=1`.

LI can be used to transfer `z=1` into `r1=z`; to obtain `r1=1`. However, this will disable the conditional write to `x`. The resulting process cannot produce `1` for `r2` or `r3` since the only writes available produce `0`.

□

EXAMPLE 23 (Example 3 revisited) Consider the program

```
x=0;y=0;(r1=x;r2=x;y=(r1==r2)?1 | r3=y;x=r3)
```

Such a program should not exhibit `r1==0,r2==1,r3==1`, since the only justification for `r3=1` appears to require `r1==r2`.

The use of DR (and hence DL) permits `r1==0;r2==1,r3==1`.

```
x=0;y=0;(r1=x;r2=x;y=(r1==r2)?1 | r3=y;x=r3)
x=0;y=0;(r1=x;r2=x;y=1 | r3=y;x=r3)          (DR)
x=0;y=0;(r1=x;y=1;r2=x; | r3=y;x=r3)        (CO*;DO)
x=0;y=0;r1=x;y=1;r3=y;x=r3;r2=x             (AU*)
r1=0;y=1;r3=1;x=1;r2=1                      (CO*;DO)
```

DW also permits the observation:

```
x=0;y=0;(r1=x;r2=x;y=(r1==r2)?1 | r3=y;x=r3)
x=0;y=0;(r1=x;r2=x;y=1;y=(r1==r2)?1 | r3=y;x=r3) (DW)
```

```
x=0;y=0;(r1=x;y=1;r2=x;y=(r1==r2)?1 | r3=y;x=r3) (CO*;DO)
x=0;y=0;r1=x;y=1;r3=y;x=r3;r2=x;y=(r1==r2)?1     (AU*)
r1=0;y=1;r3=1;x=1;r2=1                           (CO*;DO)
```

However, DO alone cannot exhibit this behavior. □

EXAMPLE 24 (Strength reduction) Consider the program:

```
x=1; (r=x;s=x;x=2*r | x=3); u=x
```

Can it yield `u=4`? Here is a derivation:

```
x=1;(r=x;s=x;x=2*r | x=3);u=x
x=1;(r=x;s=x;x=r+r | x=3);u=x   (DO, x=2*r->x=r=r)
x=1;(r=x;s=x;x=r+s | x=3);u=x   (DR)
x=1;(r=x;s=x;x=r+s | x=3);u=x   (SE, SE)
x=1;r=x;x=3;s=x;x=r+s;u=x       (AU*)
r=1;s=3;x=4;u=4                 (CO*;DO*)
```

The use of DR replaces `r=x;s=x;x=r+r` with `r=x;s=x;x=r+s`; DW and hence DO cannot accomplish this. □

5. Synchronization constructs

Synchronization constructs are defined in the RAO model by introducing extra structure to the model, and, if necessary, adding restrictions on the application of various transformations. The basic idea behind synchronization constructs is to introduce mechanisms by which the programmer may reliably communicate values from one thread to another without introducing races, i.e. the possibility of cross-overs. We illustrate by considering different flavors of *volatile* variables. The central idea of volatile variables is to provide a way by which values can be communicated from one thread to another in a "wait-free" manner (without introducing the possibility of deadlock). In Section 5.2 we treat the definition of volatiles as given by JLS2 [GJSB00]. In practice this definition has not proved to provide strong enough guarantees to be used reliably by application programmers. The central issue is the visibility of writes to raw (non-volatile) variables "through" a cross-thread volatile write/read access chain. This leads to stronger definitions of volatile discussed in subsequent sections.

5.1 Raw variables

Raw variables correspond to the "usual" (unsychronized) variables of Java. These are not associated with any special semantics for concurrent reads and writes. In particular, multiple reads and writes may be performed on the variable at the same time and a read is permitted to read the value of any *hb*-consistent write. (Thus links are permitted for such variables.) Thus raw variables can be involved in cross-overs.

5.2 JLS 2 volatiles

The informal requirement for JLS 2 volatiles is that the read of a variable x by a step s must be answered by a step t ordered before s. This can be formalized in RAO as follows. First, we distinguish between raw variables and volatile variables in the model: the underlying set V of variables is partitioned into V_r (the subset of raw variables) and V_v (the subset of volatile variables). An additional restriction is introduced on the applicability of transformations to volatile variables:

> **JLS 2 Volatility Condition:** LI may not be used to link volatile variables.

Therefore the only way to connect a write by a step s to a read by a different, unordered step t is to use AU to *hb*-order s before t, and use CO to compose the steps.

EXAMPLE 25 (Fig 21) Consider the AO process:

```
x=0; (v = 1 | r1=v; r2=v)
```

If `v` is not volatile this process may exhibit the behavior `r1==1,r2==0`:

```
s0: v=0; (s1: v=1 | s2: r1=v; s3: r2=v)
s0: v=0; (s1: v=1 | s2: r1=v;  s3: r2=1)(LI, s1->s3)
s0: v=0; (s1: v=1 | s2: r1=0; s3: r2=1) (LI, s0->s2)
```

However, if `v` is volatile the application of LI is not permitted. For `r1` to read `1`, `s1` must be ordered before `r2`. And it must lie after `s0`. But then it will force `r2=1`. □

However, JLS 2 volatiles do not guarantee reliable visibility of writes to raw variables through a volatile write/read pair.

EXAMPLE 26 (Fig 8) Consider the AO process, with `v` volatile:

```
x=0; v=false; (x=1; v=true | r1=v; r2=r1?x )
```

It is desired that if the write to `r2` executes, it writes `1`. That is, a write on a raw variable `x` can be communicated reliably through the synchronization offered by the write to the volatile variable `v`.

Unfortunately, this behavior is not guaranteed. For instance:

```
x=0; v=false; (x,v=1,true | r1=v; r2=r1?x) (CO)
x=0; v=false; (v=true; x=1 | r1=v; r2=r1?x)(SE)
x=0; v=false; (v=true; r1=v; r2=r1?x; x=1) (AU, AU)
x=0; v=false; (v,r1,r2=true,true,x; x=1)   (CO,CO,CO)
x,v,r1,r2=0,true,true,0; x=1               (CO,CO)
x,v,r1,r2=1,true,true,0                    (CO)
```

□

Examples also demonstrate that JLS 2 volatiles do not satisfy the Fundamental Property, as this example illustrates.

EXAMPLE 27 (Fig 8) Consider the AO process, with `v0,v1` volatile:

```
 v0,v1=0,0; (v0=(v0==0)?1 |
v1=(v0==1)?2 | r2=v1;r1=v0)
```

By reordering the two steps of the last thread it is possible to obtain an execution in which `r1=0` and `r2=2`. However this behavior is not an SC behavior of the original program. Since all steps involve volatile variables (or variables private to a thread) there are no races. Hence FP is not satisfied. □

5.3 DX-restricted Volatiles

The root cause of this problem is that writes to raw variables are permitted to be reordered with writes to volatile variables. This can be prevented in RAO by requiring in addition to the condition in the previous section:

> DX **Restriction:** DX may not be used to decompose f if f reads or writes a volatile variable.

EXAMPLE 28 (Fig 8, revisited) Consider the AO process, with `v` volatile:

```
x=0; v=false; (x=1; v=true | r1=v; r2=r1?x )
```

Now the desired behavior (if the write to `r2` executes, it writes `1`) can be guaranteed. The only way for `r1=v` to see `v=true` is through an AU (preceded optionally by a CO of `x=1` and `v=true`), followed by a CO. But then it must be the case that `x=1` *hb* `r2=r1?x` (or `x,v=1,true` *hb* `r2=r1?x`), and the desired behavior is guaranteed. □

Therefore the DX restriction does what it was intended to do. We note in passing that this restriction may be considered too strict.

EXAMPLE 29 Consider the AO process, with v volatile:

```
x=0;y=0;v=0;
(r1=x;r2=x;v=(r1==r2)?1; y=(r1==r2)?1
|s=y; x=s)
```

Informally, this test should permit r1=r2=1 since v is being written to and never read. Therefore if we are not interested in the value of v, the program should have the same behavior as

```
x=0;y=0;(r1=x; r2=x; y=(r1==r2)?1 | s=y; x=s )
```

which definitely permits r1=r2=1 (see Example 10). □

We remark that this problem can be addressed by moving to a slightly richer variation of RAO that we shall call RAOS. (RAO with sequencing). In RAOS, a step (such as r1,r2,v,y=x,x,1,1) is thought of as producing an *ordered sequence* of writes (instead of just a set of writes). Now there is enough extra structure to provide a more fine-grained treatment of sequencing. In RAOS no restriction is placed on DX: f can be decomposed into $g;h$ provided that for any input store f produces the same *sequence* of writes that $g;h$ does (and g and h write disjoint variables). A new transformation, RE (*reordering*) is introduced: two successive steps $g;h$ may be transformed into $h;g$ only if $h \circ g = g \circ h$ and h does not write a volatile variable. In RAOS the above example works: r1,r2,v,y=x,x,1,1 may be reduced to r1,r2,v=x,x,1;y=1 through DX.

5.4 JLS 3 volatiles

EXAMPLE 30 (Fig 22) Consider the process:

```
 x=0; y=0; v=0;
(r1=x;v=0;r2=v;y=1 | r3=y;v=0;r4=v;x=1)
```

where only the variable v is volatile. The model permits the behavior r1=r3=1 per the following derivation:

```
x=0; y=0; v=0; (r1=1;v=0;r2=v;y=1 | r3=1;v=0;r4=v;x=1)
  (LI,LI)
x=0;y=0;v=0; (r1,v,r2,y=1,0,0,1|r3,v,r4,x=1,0,0,1)
  (CO*)
```

□

The resulting process is a completed execution, with a cross-over. Note that all the reads of the volatile variable v have not been totally ordered in the above example. The JLS 3[GJSB05] design for volatiles solves this problem by requiring a a total *synchronization order* (SO) on all reads and writes of volatile variable x. Further, there is required to be an *hb*-edge between a write of a volatile variable x and *all* SO-subsequent reads of x.

Formally, this requirement is implemented in RAO exactly as stated above. In addition to the requirements of the previous two sections, we redefine the notion of completed execution as follows:

> **JLS 3 Volatility Condition:** An AO process is a completed execution iff all its steps are completed and there exists a total order on all steps that read or write volatile variables (the *synchronization order*, SO) and there is an *hb*-edge between a write of a volatile variable x and all SO-subsequent reads of x.

Modulo this change, the notion of SC execution is unchanged from Definition 9. To satisfy this requirement *hb*-edges may need to be added, using AU. [7] With these conditions all three Test Cases – (25, 26 and 30) – are satisfied.

EXAMPLE 31 (Fig 22, revisited) Consider the process:

```
x=0; y=0; v=0; (r1=x;v=0;r2=v;y=1 | r3=y;v=0;r4=v;x=1)
```

where only v is volatile. Consider:

```
 x=0;y=0;v=0;
 (s0: r1=x; v1: v=0; v2: r2=v; s1: y=1
  | s2: r3=y; v3: v=0; v4: r4=v; s3: x=1)
x=0;y=0;v=0;
 (s0: r1=1; v1: v=0; v2: r2=v; s1: y=1
  | s2: r3=1; v3: v=0; v4: r4=v; s3: x=1)
          (LI s3->s0,s1->s2)
```

The resulting process is not a completed execution. There must be a total synchronization order on the steps v1,v2,v3,v4 satisfying the desired condition. Either v2 must lie after v3 or v4 must lie after v1. Any attempts to add *hb*-edges to satisfy the condition above will result in the conditions for one of the links to be violated: the target of a link will be *hb* its source. Therefore it is not possible to complete this process. □

We remark in passing that the JLS 3 notion of volatiles can be strengthened by requiring that a completed execution must satisfy the condition that there is an *hb* edge between a write of *any* volatile variable and SO-subsequent reads of *any* volatile variable. The definition of this variant – call it JLS3-ALL – on top of RAO is straightforward.

5.5 Totally ordered volatiles (TOV)

The above design strengthens the connection between a volatile write and subsequent reads. It does not totally order the writes. Thus cross-overs involving volatile reads and writes are still possible.

EXAMPLE 32 (Fig 22a) Consider the process obtained by removing the second volatile read r4=v:

```
 x=0; y=0; v=0;
(r1=x;v=0;r2=v;y=1 | r3=y;v=0;x=1)
```

Again, only the variable v is JLS 3 volatile. This program has the behavior r1=r3=1:

```
 x=0;y=0;v=0;
 (s0: r1=x; v1: v=0; v2: r2=v; s1: y=1
  | s2: r3=y; v3: v=0; s3: x=1)
x=0;y=0;v=0;
 (s0: r1=1; v1: v=0; v2: r2=v; s1: y=1
  | s2: r3=1; v3: v=0; s3: x=1)  (LI s3->s0,s1->s2)
```

The resulting process *is* a completed execution – with the synchronization order v3,v1,v2. It has a cross-over {s0,v1,v2,s1,s2,v3,s3} involving steps of two threads, with actions on the same volatile variable from each of those threads. By removing the second volatile read, the additional restriction of the previous section was taken out of play. □

If to the previous three volatility condition we add:

> **Volatility Total Order Condition:** An AO process is a completed execution if all steps are completed and all steps

[7] The addition of AU edges may not be possible because of the presence of links. Thus it is possible that starting with an AO process P, there is a sequence of linkings resulting in a process which cannot be completed into an execution. A safe strategy is to first introduce AU edges as needed to satisfy the condition above, and then add LI links.

that read and write a volatile variable x are totally ordered by *hb*.

then Fig 22a would no longer have a valid execution. In a completed execution, we must have either `v3 hb v1` or `v1 hb v3`. But AU cannot be used to estabish these edges for the last process in the example above because in either case the wb_X condition associated with one of the links will not be satisfied.

More generally, with this condition, there cannot be a cross-over c which contains a volatile action on the same variable v for each thread that has steps in c. We illustrate the case for a set of actions belonging to two threads. Consider an execution. Let s be the first volatile step in c in that execution. Say it belongs to thread t_0. Let t be the first step in the c after s in the execution and belonging to the other thread, t_1. For s,t to be in a cross-over, there must be a link from a step after t to a step before s. However, every step in t_1 after t happens after every step in t_0 before s (because of the Volatility Total Order Condition) and so cannot be used as a source for a link into a step in t_0.

As with JLS3-ALL we remark that a variant TOV-ALL can be defined on top of RAO. It requires that a completed execution must totally order all read and write steps on all volatile variables.

5.6 Main theorem

We now consider the formal properties of the RAO model, extended with the definitions of volatile variables discussed above. Let Vl range over the definitions of volatiles (excluding JLS 2, which does not satisfy the Fundamental Property, as discussed above). Let the notion of an RAO(Vl) (AO(Vl)) model stand for the notion of an RAO (AO) model on top of a set of variables which are partitioned into raw and volatile variables, and for which the application of transformations on volatile variables is restricted per Vl, and the definition of completed execution is changed (if necessary) as per Vl. The following results carry over from AO.

LEMMA 27 (GOOD BEHAVIOR IS $\xrightarrow{X}$-INVARIANT.). *For all* AO(Vl) *processes P,Q if P is well-behaved and $P \xrightarrow{X} Q$ then Q is well-behaved.*

The proof rests on the proof of the corresponding theorem for AO, and the simple observation that volatile variables cannot introduce races (for each definition of volatile).

LEMMA 28 (PRESERVATION OF SC BEHAVIOR). *Let P,Q be* AO(Vl) *processes such that $P \xrightarrow{X} Q$, and P is well-behaved. Then* $\mathrm{sc}(Q) \subseteq \mathrm{sc}(P)$.

Because of interactions with links, Q may no longer have an SC execution. The lemma then follows vacuously. Otherwise, the proof rests on transforming the total order underlying an SC execution of P into a total order on the steps of Q. The cases are straightforward.

The proofs of the main properties rest on the corresponding ones for AO.

LEMMA 29 (MAIN LEMMA). *For all* AO(Vl) *processes P,Q if P is well-behaved and $P \xrightarrow{X} Q$ then* $\mathrm{io}(Q) \subseteq \mathrm{sc}(P)$.

THEOREM 30 (FUNDAMENTAL PROPERTY). *Let P be a well-behaved* AO(Vl) *process. Then* $\mathrm{io}(P) \subseteq \mathrm{sc}(P)$.

THEOREM 31 (FUNDAMENTAL PROPERTY). *Let P be a well-behaved* AO(Vl) *process. Then* $\mathrm{io}(P) \subseteq \mathrm{sc}(P)$.

6. Related work

Location consistency model. Location consistency (LC) [GS00] is probably the weakest memory model described in the literature. The distinguishing property of LC is that it does not rely on coherence, thus dispensing the need for cache snooping and directories in a multiprocessor implementation. Gao and Sarkar argue that the model is equivalent to release consistency (RC) [GLL+90] for programs that are data race free. However, unlike RAO the specification of LC is not suited as a basis for a memory model of a high-level programming language as it does not explicitly define which re-orderings of access and synchronization statements are permitted [WTA02]. Like LC, RAO does not rely on the coherence assumption.

OpenMP and UPC memory models. The memory models of OpenMP [HdS05] and UPC [YBW04] have been specified after the original specification of these language extensions. The fundamental difference with RAO is as follows: Both OpenMP and UPC are founded on programming languages with unsafe typing and pointer arithmetic and thus the requirements that their memory models impose on programs that are *not* data race free can be looser. RAO, in contrast, is designed for type safe-languages like X10 or Java with the strong memory safety in mind. The focus of the specification of the UPC and OpenMP memory model is on the effect and ordering guarantees provided by certain accesses with synchronization semantics and explicit synchronization constructs – not on guarantees that are given in the absences of such synchronization. Both models allow the introduction of spurious writes, and reads may observe "out of thin air" values in programs with data races [Boe05].

Java memory model. The RAO model can be thought of as a "happens before" model, discussed in [MPA05, Section3]. RAO is generative, given a source program it generates all possible sequences of executions. In contrast, the methodological stance of [MPA05] is that a trace must be given beforehand; the memory model is then specified in terms of which traces are correct. We feel that valuable information is lost when one moves from a generative model to an oracle; in particular, the task of specifying the semantics is made harder.

ASIAN 2004 paper. This paper generalizes and simplifies [Sar04]. The core concept of linking is derived from the action sets of [Sar04]. The "unique valuation" condition has been replaced by the simpler well-foundedness condition. Conditional linkings have been done away with in favor of (partial) steps. The formulation of the model in terms of a set of permitted transformations is new to this paper.

7. Conclusion and future work

We believe this paper is a first step towards establishing a systematic understanding of weak memory models.

Important foundational questions remain. Is RAO the "weakest" relaxed memory model satisfying the requirements laid out in Section 1? This hinges on formalizing the "No Thin Air Reads" condition. What compositionality properties – and hence reasoning principles – are satisfied by RAO? We believe deeper results may be obtained, particularly by connecting with ideas from separation logic [Bro04]. An efficient implementation of the calculus presented in this paper would be valuable. Further, we believe that it is valuable to develop (and mechanize) the ideas of RAO for *concurrent data structures*, that offer richer semantic functions than reads/write operations on memory locations, in the spirit of [BAM06].

On this foundation several synchronization constructs can be defined. In particular, a systematic account can be given of locks,

and isolated and atomic execution, in the context of transactional memory. These ideas will be developed in subsequent work.

Acknowledgements We gratefully acknowledge extended discussions with Doug Lea, Bill Pugh, Jeremy Manson, Allan Kielstra, Vivek Sarkar, Suresh Jagannathan, Tony Hoare, and the participants of the Java Memory Model list. Vijay Saraswat, Maged Michael and Christoph von Praun were supported in part by DARPA under contract No. NBCH30390004. Radha Jagadeesan was supported in part by NSF 0430175.

References

[BAM06] S. Burckhardt, R. Alur, and M.M.K. Martin. Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study. In *CAV*, 2006.

[Boe05] Hans-Juergen Boehm. Threads cannot be implemented as a library. In *PLDI*, pages 261–268, 2005.

[Bro04] S.D. Brookes. A Semantics for Concurrent Separation Logic. In *Concur*, pages 16–34, 2004.

[CDG+05] Philippe Charles, Christopher Donawa, Christian Grothoff, Kemal Ebcioglu, Allen Kielstra, Vijay Saraswat, Vivek Sarkar, and Christoph von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[Cur93] P-L. Curien. *Categorical Combinators, Sequential Algorithms andFunctional Programming*. Birkhauser, second edition, 1993.

[GJSB00] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[GJSB05] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2005.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'90)*, pages 15–26, June 1990.

[GS00] G. Gao and V. Sarkar. Location Consistency – A New Memory Model and Cache Consistency Protocol. *IEEE Transactions on Computers*, 49(8):798–813, August 2000.

[HdS05] J. Hoeflinger and B. de Supinsky. The openmp memory model. June 2005.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[MPA05] Jeremy Manson, Bill Pugh, and Sarita Adve. The Java Memory Model. In *POPL '05. Proceedings of the 32d ACM SIGPLAN-SIGACT on Principles of programming languages*, January 2005.

[Pug04] W. Pugh. Java Memory Model Causality Test Cases. Technical report, U Maryland, 2004. On `www.cs.umd.edu`, as `~pugh/java/memoryModel/`.

[Sar04] Vijay Saraswat. Concurrent Constraint-Based Memory Machines: A Framework for Java Memory Models. In *ASIAN*, pages 494–508, 2004.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David A. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPOPP*, pages 2–13, 2005.

[SJ05] Vijay Saraswat and Radha Jagadeesan. Concurrent Clustered Programming. In *Concur*, pages 353–367, 2005.

[SS88] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programsthat Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[Vui74] Jean Etienne Vuillemin. *Proof-techniques for recursive programs.* PhD thesis, 1974.

[WTA02] C. Wallace, G. Tremblay, and J. Amaral. the tamability of the location consistency memory model, 2002.

[YBW04] K. Yelick, D. Bonachea, and Ch. Wallace. A Proposal for a UPC Memory Consistency Model, v1.1. Technical Report LBNL Technical Report (draft), 2004.

A. Mathematical details

PROPOSITION 32. *$c[c[d]] = c[d]$, and $c[d[e]] = c[d][e]$, for any stores c,d,e.*

PROPOSITION 33. *$c[d] \geq d$, for any stores c,d.*

PROPOSITION 34. *$d \geq e$ implies $c[d] \geq c[e]$, for any stores c,d,e.*

In fact, Propositions 32–34 establish that for all c, $c[\_]$ is a closure operator.

Let P,Q be AO processes. Say that $P \xrightarrow{X} Q$ if Q is obtained from P by the application of one of the RAO transformations. The *SC i/o functions* of P, $sc(P)$ is the set of functions $io(s_0 \circ \ldots \circ s_{n-1})$ where $\{s_0,\ldots,s_{n-1}\}$ is a totally ordered extension of P (with steps enumerated in *hb*-order). (Note that $sc(P)$ may be empty because no totally ordered extension of P may exist, because of volatility conditions.)

Let $clo(P)$ represent the set of AO processes obtained from P by zero or more applications of the given transformations. Say that P is *complete* if for every step $p \in P$, $\in (p) \in dom(p)$. We take the *observations* of a process P to be the set of *i/o functions* of P, $io(P)$ defined as the set $\{f \in sc(Q) \mid Q \in clo(P), Q \text{ complete}\}$. We say that $O \in io(P)$ has *a proof of size n* if there is an $\xrightarrow{X}$ sequence of length n from P to Q such that $O \in sc(Q)$.

Say that an AO process P is *well-behaved* if all its SC executions are race-free.

LEMMA 35 (GOOD BEHAVIOR IS $\xrightarrow{X}$-INVARIANT.). *For all AO processes P,Q if P is well-behaved and $P \xrightarrow{X} Q$ then Q is well-behaved.*

Proof. By inspection of the proof rules.

Let $X = \mathsf{IM}$, with $f \in P$ replaced by its improvement $g \in Q$. Note that if g has a race with some step e then in the same sequence (with g replaced by f) of the steps of P there is a race between f and e. Therefore no total order of the steps of Q has a race.

If $X = \mathsf{CO}$, with f and g being replaced by h, then any total order of the steps of Q corresponds to the same total order on the steps of P with h being replaced by f and g in succession. The sequence for Q has a race iff the sequence for P does.

Let $X = \mathsf{DL}$, and $f \in P$ be replaced by $g,h \in Q$. Consider a total ordering Z of the steps of Q. We need to consider two cases. First, suppose there is a race between g and a step e. From Z form a total order Z' of the steps of P with g replaced by f, and h by nothing. Then there will be a race between f and e in Z'. But there cannot be. Now consider the second case: there is a race in Z between two steps that lie after g in Z. If not, find the first step e after g in Z whose input store (confined to the variables it reads) is different from that for e in Z'. Then e and f are in a race in Z'. But this cannot be. Hence Z has no races.

Let $X = \mathsf{LI}$ with $g \in P$ replaced by $f \rightarrow_V g$. Since P has no races, and f and g are unordered in P, it follows that $f \rightarrow_V g = g$. (If not,

then one can find a total order of the steps of P in which f precedes g and now g will read some write of f, thereby exhibiting a race.) Therefore every total order of the steps of Q is in fact a total order of the steps of P. Since no total order of the steps of P has a race, neither does any total order of the steps of Q.

Let $X = \mathsf{PR}$, with f replaced by a Z-equivalent step q. Every total order of Q corresponds to an identical total order of the steps of P, with g replaced by f. There is a race in the total order of Q iff there is a race in the corresponding total order of P.

Let $X = \mathsf{AU}$. The total orders of the steps of Q are a subset of those of P. Since the total orders of the steps of P have no races, we are done.

LEMMA 36 (PRESERVATION OF SC BEHAVIOR). *Let P,Q be* AO *processes such that $P \xrightarrow{X} Q$, and P is well-behaved. Then* $\mathrm{sc}(Q) \subseteq \mathrm{sc}(P)$.

Proof. $X = \mathsf{IM}$. A step f in g is replaced by a step g which improves f. But $io(g) = io(f)$ (precondition for IM), and we know that $io(f;g) = io(f);io(g)$. Therefore the total order of the steps in Q that witnesses the given observation can be modified (by replacing g by f) to obtain the total order of the steps in P that exhibit the same observation.

$X = \mathsf{CO}$. Let $f,g \in P$ be changed to $h = f \circ g$ in Q. Let $\{s_0,\ldots,s_{n-1}\}$ be the total order of the steps in Q that yields the desired observation. Replace h in this order by f followed by g to get a total order of P which has the same observation.

$X = \mathsf{DL}$. Let $f \in P$ be replaced by g,h in Q. Since P is well-behaved, by the previous lemma Q is well-behaved. Now any SC execution Z of Q corresponds to an SC execution Z' of P in which the steps g and h have been removed and g has been replaced by f. (The next step after g in Z that is sensitive to the output of g has to be h or some successor step, since no SC execution of Q has races.)

$X = \mathsf{LI}(f,g,x,v)$. There are two cases. A step $f \in P$ is used to replace an unordered step $g \in P$ by $h = f :_{\{x\}} g$ to get the procss P'. But note that since there are no races in P it must be the case that f cannot produce any information that g can read, i.e. $h = g$. Hence the total order on Q that witnesses the given observation is also the total order of steps of P. Alternatively, f *hb* g. Then it follows that there is no other step g' such that f *hb* g' *hb* g. Let Z be a total order on the steps of P'. Let Z' be Z with h replaced by g. Then Z' is a total order of the steps of P and the input store i_g before g in Z is the same as the input store i_h before h in Z'. Note that $g(i_g) = h(i_h)$. Therefore the input store before every step in Z is the same as the input store before the corresponding step in Z'. Therefore the io function of Z and Z' are the same.

$X = \mathsf{PR}$. A step $f \in P$ is replaced with a step $g \in Q$ such that f and g are Z-equivalent, where Z is a constraint that holds in all SC executions of P before f. Consider a total order of the steps of Q that witnesses the given observation. Let g be the ith element in the sequence. Then all the steps before i are in common with P. It must be the case that Z is true. Since f and g are Z-equivalent we can replace g by f and keep the rest of the sequence unchanged to get a sequence that is a total ordering of the steps of P and has the same io function.

$X = \mathsf{AU}$. An extra *hb* -edge is added to get Q. Now any total order of Q is a total order of P, so we are done.

LEMMA 37 (MAIN LEMMA). *For all* AO *processes P,Q if P is well-behaved and $P \xrightarrow{X} Q$ then* $\mathrm{io}(Q) \subseteq \mathrm{sc}(P)$.

Proof. By induction on the size of the proof of an observation $O \in io(Q)$,

The inductive hypothesis is:

For all AO processes P and Q and observations O, if $P \xrightarrow{X} Q$, P is well-behaved, $O \in io(Q)$ has a proof of size n then $O \in sc(P)$.

Base case ($n = 0$): In this case $P = Q$, $O \in sc(Q)$. So $O \in sc(P)$.

Inductive case ($n > 0$). Assume the inductive hypothesis for n. We will establish it for $n+1$. Assume P is well-behaved, $P \xrightarrow{X} Q$, $O \in io(Q)$ has a proof of size $n+1$. We have to show $O \in sc(P)$.

If O has a proof of size $n+1$, then for some Q', $Q \xrightarrow{Y} Q'$ and $O \in io(Q')$ with a proof of size n. By Lemma 35, Q is well-behaved. So by I.H. $O \in sc(Q)$. By Lemma 36, $O \in sc(P)$.

THEOREM 38 (FUNDAMENTAL PROPERTY). *Let P be a well-behaved* AO *process. Then* $\mathrm{io}(P) \subseteq \mathrm{sc}(P)$.