Abstraction for Concurrent Objects*

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang

Queen Mary University of London, UK

Abstract. Concurrent data structures are usually designed to satisfy correctness conditions such as sequential consistency and linearizability. In this paper, we consider the following fundamental question: what guarantees are provided by these conditions for client programs? We formally show that these conditions can be *characterized* in terms of observational refinement. Our study also provides a new understanding of sequential consistency and linearizability in terms of abstraction of dependency between computation steps of client programs.

1 Introduction

The design and implementation of correct and efficient concurrent programs is a challenging problem. Thus, it is not surprising that programmers prefer to develop concurrent software mainly by utilizing highly-optimized concurrent data structures that have been implemented by experts.

Unfortunately, there is a gap in our theoretical understanding, which can have a serious consequence on the correctness of client programs of those concurrent data structures. Usually, programmers expect that the behavior of their program does not change whether they use experts' data structures or less-optimized but obviously-correct data structures. In the programming language community, this expectation has been formalized as observational refinement [4,8,11]. On the other hand, concurrent data structures are designed with different correctness conditions proposed by the concurrent-algorithm community, such as *sequential consistency* [9] and *linearizability* [6]. Can these correctness conditions meet programmers' expectation? In other words, what are the relationships between these conditions and observational refinement? As far as we know, no systematic studies have been done to answer this question.

The goal of this paper is to close the aforementioned gap. We show that (1) linearizability coincides with observational refinement, and (2) as long as the threads are non-interfering (except through experts' concurrent data structures), sequential consistency is equivalent to observational refinement. Our results pinpoint when it is possible to replace a concurrent data structure by another sequentially consistent or linearizable data structure in (client) programs, while preserving observable properties of the programs. One direction in this connection (that linearizability implies observational refinement) has been folklore amongst concurrent-algorithm researchers, and our results provide the first formal confirmation of this folklore. On the other hand, as far as we are aware the other direction (when observational refinement implies linearizability or sequential consistency) is not prefigured or otherwise suggested in the literature.

* We would like to thank anonymous referees, Viktor Vafeiadis and Matthew Parkinson for useful comments. This work was supported by EPSRC.

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 252–266, 2009.

Programs, Object Systems, and Histories. A concurrent data structure provides a set of procedures, which may be invoked by concurrently executing threads of the client program using the data structure. Thus, procedure invocations may overlap. (In our setting, a data structure can neither create threads nor call a procedure of the client.) We refer to a collection of concurrent data structures as an *object system*.

In this paper, we are not interested in the implementation of an object system; we are only interested in the possible interactions between the client program and the object system. Thus, we assume that an object system is represented by a set of *histories*. Every history records a possible interaction between the client application program and the object system. The interaction is given in the form of sequences of procedure invocations made by the client and the responses which it receives. A program can use an object system only by interacting with it according to one of the object system's histories. [1]

Example 1. The history $H_0 = (t_1, \mathit{call}\ q.enq(1)); (t_1, \mathit{ret}()\ q.enq); (t_2, \mathit{call}\ q.deq()); (t_2, \mathit{ret}(1)\ q.deq)$ records an interaction in which thread t_1 enqueues 1 into queue q followed by a dequeue by thread t_2. The histories

$$H_1 = (t_1, \mathit{call}\ q.enq(1))(t_1, \mathit{ret}()\ q.enq)(t_2, \mathit{call}\ q.enq(2))(t_2, \mathit{ret}()\ q.enq)$$
$$H_2 = (t_2, \mathit{call}\ q.enq(2))(t_2, \mathit{ret}()\ q.enq)(t_1, \mathit{call}\ q.enq(1))(t_1, \mathit{ret}()\ q.enq)$$
$$H_3 = (t_1, \mathit{call}\ q.enq(1))(t_2, \mathit{call}\ q.enq(2))(t_1, \mathit{ret}()\ q.enq)(t_2, \mathit{ret}()\ q.enq)$$

record interactions in which thread t_1 enqueues 1 into the queue and thread t_2 enqueues 2. In H_1, the invocation made by t_1 happens before that of t_2 (i.e., t_1 gets a response before t_2 invokes its own procedure). In H_2, it is the other way around. In H_3, the two invocations overlap.

Sequential Consistency and Linearizability. Informally, an object system OS_C is *sequentially consistent* wrt. an object system OS_A if for every history H_C in OS_C, there exists a history H_A in OS_A that is just another interleaving of threads' actions in H_C: in both H_C and H_A, the same threads invoke the same sequences of operations (i.e., procedure invocations) and receive the same sequences of responses. We say that such H_C and H_A are *weakly equivalent*. (We use the term *weak* equivalence to emphasis that the only relation between H_C and H_A is that they are different interleavings of the same sequential threads.) OS_C is *linearizable* wrt. OS_A, if for every history H_C in OS_C, there is some H_A in OS_A such that (1) H_C and H_A are weakly equivalent and (2) the global order of non-overlapping invocations of H_C is preserved in H_A.[2] In the context of this paper, the main difference between sequential consistency and linearizability is, intuitively, that the former preserves only the happens-before relation between operations of the *same* thread while the latter preserves this relation between the operations of *all* threads.

[1] This is a standard assumption in concurrent algorithms work, which Herlihy and Shavit refer to as *interference freedom* [5]: it is an assumption which would have to be verified by other means when applying the theory to particular programming languages or programs.

[2] It is common to require that OS_A be comprised of sequential histories, i.e., ones in which invocations do not overlap. (In this setting, linearizability intuitively means that every operation appears to happen atomically between its invocation and its response.) However, this requirement is not technically necessary for our results, so we do not impose it.

Example 2. The histories H_1, H_2, and H_3 are weakly equivalent. None of them is weakly equivalent to H_0. The history H_3 is linearizable wrt. H_1 as well as H_2, because H_3 does not have non-overlapping invocations. On the other hand, H_1 is not linearizable with respect to H_2; in H_1, the enqueue of t_1 is completed before that of t_2 even starts, but this global order on these two enqueues is reversed in H_2.

Observational Refinement. Our notion of observational refinement is based on observing the initial and final values of variables of client programs. (One can think of the program as having a final command "print all variables".) We say that an object system OS_C observationally refines an object system OS_A if every program P with OS_A, replacing OS_A by OS_C does not generate new observations: for every initial state s, the execution of P with OS_C at s produces only those output states that can already be obtained by running P with OS_A at s.

The main results of this paper is the following characterization of sequential consistency and linearizability in terms of observational refinement:

1. OS_C observationally refines OS_A iff OS_C is sequential consistent with respect to OS_A, assuming client operations (e.g., assignments to variables) of each thread access thread-local variables (or resources) only.
2. OS_C observationally refines OS_A iff OS_C is linearizable with respect to OS_A, assuming that client operations may use at least one shared global variable.

We start the paper by defining a programming language and giving its semantics together with the formal definition of observational refinement (Sections 2, 3, 4 and 5). Then, we describe a generic technique for proving observational refinement in Section 6, and use this technique to prove the connection between observational refinement and linearizability or sequential consistency in Section 7. The next section revisits the definitions of sequential consistency and linearizability, and provides the analysis of them in terms of the dependency between computation steps. Finally, we conclude the paper in Section 9. For space reasons, some proofs are omitted. They can be found in the full version of the paper [3].

2 Programming Language

We assume that we are given a fixed collection O of objects, with method calls $o.f(n)$. For simplicity, all methods will take one integer argument and return an integer value. We will denote method calls by $x{:=}o.f(e)$.

The syntax of sequential commands C and complete programs P is given below:

$$C \ ::= \ \mathtt{c} \mid x{:=}o.f(e) \mid C;C \mid C + C \mid C^\star \qquad P \ ::= \ C_1 \parallel \cdots \parallel C_n$$

Here, $\mathtt{c}$ ranges over an unspecified collection $\mathtt{PComm}$ of primitive commands, $+$ is nondeterministic choice, ; is sequential composition, and $(\cdot)^\star$ is Kleene-star (iterated ;). We use $+$ and $(\cdot)^\star$ instead of conditionals and while loops for theoretical simplicity: given appropriate primitive actions the conditionals and loops can be encoded. In this paper, we assume that the primitive commands include assume statements $\mathtt{assume}(b)$ and assignments $x{:=}e$ not involving method calls.[3]

[3] The $\mathtt{assume}(b)$ statement acts as skip when the input state satisfies b. If b does not hold in the input state, the statement deadlocks and does not produce any output states.

3 Action Trace Model

Following Brookes [2], we will define the semantics of our language in two stages. In the first there will be a trace model, where the traces are built from atomic actions. This model resolves all concurrency by interleaving. In the second stage, which is shown in Section 5, we will define the evaluation of these action traces with initial states.

Definition 1. *An* **atomic action** *(in short, action) φ is a client operation or a call or return action: $\varphi ::= (t, a) \mid (t, call\, o.f(n)) \mid (t, ret(n)\, o.f)$. Here, t is a thread-id (i.e., a natural number), a in (t, a) is an atomic client operation taken from an unspecified set Cop_t (parameterized by the thread-id t), and n is an integer. An* **action trace** *(in short, trace) τ is a finite sequential composition of actions (i.e., $\tau ::= \varphi; \cdots; \varphi$).*

We identify a special class of traces where calls to object methods run sequentially.

Definition 2. *A trace τ is* **sequential** *when all calls in τ are immediately followed by matching returns, that is, τ belongs to the set*

$$\Big(\bigcup_{t,a,o,f,n,m} \{\, (t,a),\ \ (t, call\, o.f(n)); (t, ret(m)\, o.f) \,\} \Big)^* \Big(\bigcup_{t,o,f,n} \{\, \epsilon,\ \ (t, call\, o.f(n)) \,\} \Big).$$

Intuitively, the sequentiality means that all method calls to objects run atomically. Note that the sequentiality also ensures that method calls and returns are properly matched (possibly except the last call), so that, for instance, no sequential traces start with a return action, such as $(t, ret(3)\, o.f)$.

The execution of a program in this paper generates only well-formed traces.

Definition 3. *A trace τ is* **well-formed** *iff for all thread-ids t, the projection of τ to the t-thread, $\tau|_t$, is sequential.*

The well-formedness formalizes two properties of traces. Firstly, it ensures that all the returns should have corresponding method calls. Secondly, it formalizes the intuition that each thread is a sequential program, if it is considered in isolation. Thus, when the thread calls a method $o.f$, it has to wait until the method returns, before doing anything else. We denote the set of all well-formed traces by $WTraces$.

Our trace model $T(-)$ defines the meaning of sequential commands and programs in terms of traces, and it is shown in Figure 1. In our model, a sequential command C means a set $T(C)t$ of well-formed traces, which is parametrized by the id t of a thread running the command. The semantics of a complete program (a parallel composition) P, on the other hand, is a non-parametrized set $T(P)$ of well-formed traces; instead of taking thread-ids as parameters, $T(P)$ creates thread-ids.

Two cases of our semantics are slightly unusual and need further explanations. The first case is the primitive commands $\mathtt{c}$. In this case, the semantics assumes that we are given an interpretation $[\![\mathtt{c}]\!]_t$ of $\mathtt{c}$, where $\mathtt{c}$ means finite sequences of atomic client operations (i.e., $[\![\mathtt{c}]\!]_t \subseteq Cop_t^+$). By allowing sequences of length 2 or more, this assumed interpretation allows the possibility that $\mathtt{c}$ is not atomic, but implemented by a sequence of atomic operations. The second case is method calls. Here the semantics distinguishes calls and returns to objects, to be able to account for concurrency (overlapping operations). Given $x{:=}o.f(e)$, the semantics non-deterministically chooses two

$$T(\mathtt{c})t = \{\ (t,a_1);(t,a_2);\ldots;(t,a_k)\ \mid\ a_1;a_2;\ldots;a_k \in [\![\mathtt{c}]\!]_t\ \}$$
$$T(x{:=}o.f(e))t = \{\ \tau;(t, call\ o.f(n));(t, ret(n')\ o.f);\tau'\ \mid\ n,n' \in Integers\ \wedge\ \tau \in T(\mathtt{assume}(e{=}n))t\ \wedge\ \tau' \in T(x{:=}n')t\ \}$$
$$T(C_1;C_2)t = \{\,\tau_1;\tau_2 \mid \tau_i \in T(C_i)t\,\}\quad T(C_1{+}C_2)t = T(C_1)t \cup T(C_2)t\quad T(C^\star)t = (T(C)t)^\star$$
$$T(C_1 \parallel \cdots \parallel C_n) = \bigcup\{\ interleave(\tau_1,...,\tau_n)\ \mid\ \tau_i \in T(C_i)i\ \wedge\ 1 \le i \le n\ \}$$

Fig. 1. Action Trace Model. Here $\tau \in interleave(\tau_1,...,\tau_n)$ iff every action in τ is done by a thread $1 \le i \le n$ and $\tau|_i = \tau_i$ for every such thread i.

integers n, n', and uses them to describe a call with input n and a return with result n'. In order to ensure that the argument e evaluates to n, the semantics inserts the assume statement $\mathtt{assume}(e{=}n)$ before the call action, and to ensure that x gets the return value n', it adds the assignment $x{:=}n'$ after the return action. Note that some of the choices here might not be feasible; for instance, the chosen n may not be the value of the parameter expression e when the call action is invoked, or the concurrent object never returns n' when called with n. The next evaluation stage of our semantics will filter out all these infeasible call/return pairs.

Lemma 1. *For all sequential commands C, programs P and thread-ids t, both $T(C)t$ and $T(P)$ contain only well-formed traces.*

4 Object Systems

The semantics of objects is given using histories, which are sequences of calls and returns to objects. We first define precisely what the individual elements in the histories are.

Definition 4. *An* **object action** *is a call or return: $\psi ::= (t, call\ o.f(n)) \mid (t, ret(n)\ o.f)$. A* **history** *$H$ is a finite sequence of object actions (i.e., $H ::= \psi;\psi;\ldots;\psi$). If a history H is well-formed when viewed as a trace, we say that H is* **well-formed**.

Note that in contrast to traces, histories do not include atomic client operations (t, a). We will use $\mathcal{A}$ for the set of all actions, $\mathcal{A}_o$ for the set of all object actions, and $\mathcal{A}_c$ for $\mathcal{A} - \mathcal{A}_o$, i.e., the set of all client operations.

We follow Herlihy and Wing's approach [6], and define object systems.

Definition 5. *An* **object system** *OS is a set of well-formed histories.*

Notice that OS is a collective notion, defined for all objects together rather than for them independently. Sometimes, the traces of a system satisfy special properties.

Definition 6. *Let OS be an object system. We say that OS is* **sequential** *iff it contains only sequential traces; OS is* **local** *iff for any well-formed history H, $H \in OS \iff (\forall o.\, H|_o \in OS)$.*

A local object system is one in which the set of histories for all the objects together is determined by the set of histories for each object individually. Intuitively, locality means that objects can be specified in isolation. Sequential and local object systems are commonly used as specifications for concurrent objects in the work on concurrent algorithms. (See, e.g., [5]).

5 Semantics of Programs

We move on to the second stage of our semantics, which defines the evaluation of traces. Suppose we are given a trace τ and an initial state s, which is a function from variables $x, y, z, \ldots$ to integers.[4] The second stage is the evaluation of the trace τ with s, and it is formally described by the evaluation function eval below:

$$\mathsf{eval} \;:\; \mathit{States} \times \mathit{WTraces} \to \mathcal{P}(\mathit{States})$$
$$\mathsf{eval}(s, (t, \mathit{call}\ o.f(n)); \tau) = \mathsf{eval}(s, \tau) \qquad \mathsf{eval}(s, (t, \mathit{ret}(n)\ o.f); \tau) = \mathsf{eval}(s, \tau)$$
$$\mathsf{eval}(s, (t, a); \tau) = \bigcup_{(s,s') \in [\![a]\!]} \mathsf{eval}(s', \tau) \qquad \mathsf{eval}(s, \epsilon) = \{s\}$$

The semantic clause for atomic client operations (t, a) assumes that we already have an interpretation $[\![a]\!]$ where a means a binary relation on States. Note that a state s does not change during method calls and returns. This is because firstly, in the evaluation map, a state describes the values of client variables only, not the internal status of objects and secondly, the assignment of a return value n to a variable x in $x{:=}o.f(e)$ is handled by a separate client operation; see the definition of $T(x{:=}o.f(e))$ in Figure 1.

Now we combine the two stages, and give the semantics of programs P. Given a specific object system OS, the formal semantics $[\![P]\!](OS)$ is defined as follows:

$$[\![P]\!](OS) \;:\; \mathit{States} \to \mathcal{P}(\mathit{States})$$
$$[\![P]\!](OS)(s) \;=\; \bigcup\{\, \mathsf{eval}(s, \tau) \;\mid\; \tau \in T(P) \wedge \mathsf{getHistory}(\tau) \in OS \,\}$$

Here $\mathsf{getHistory}(\tau)$ is the projection of τ to object actions. The semantics first calculates all traces $T(P)$ for τ, and then selects only those traces whose interactions with objects can be implemented by OS. Finally, the semantics runs all the selected traces with the initial state s.

Our semantics observes the initial and final values of variables in threads, and ignores the object histories. One can think of the program as having a final command "print all variables", which gives us our observable. We use this notion of observation and compare two different object systems OS_A and OS_C.

Definition 7. *Let OS_A and OS_C be object systems. We say that*

- OS_C **observationally refines** $OS_A \iff \forall P, s.\ [\![P]\!](OS_C)(s) \subseteq [\![P]\!](OS_A)(s)$;
- OS_C *is* **observationally equivalent** *to* $OS_A \iff \forall P.\ [\![P]\!](OS_C) = [\![P]\!](OS_A)$.

Usually, OS_A is a sequential local object system that serves as a specification, and OS_C is a concurrent object system representing the implementation. The observational

[4] All the results of the paper except the completeness can be developed without assuming any specific form of s. Here we do not take this general approach, to avoid being too abstract.

refinement means that we can replace OS_A by OS_C in any programs without introducing new behaviors of those programs, and gives a sense that OS_C is a correct implementation of OS_A.

In the remainder of this paper, we will focus on answering the question: how do correctness conditions on concurrent objects, such as linearizability, relate to observational refinement?

6 Simulation Relations on Histories

We start by describing a general method for proving observational refinement. Later, in Section 7, we will show that both linearizability and sequential consistency can be understood as specific instances of this method.

Roughly speaking, our method works as follows. Suppose that we want to prove that OS_C observationally refines OS_A. According to our method, we first need to choose a binary relation $\mathcal{R}$ on histories. This relation has to be a *simulation*, i.e., a relation that satisfies a specific requirement, which we will describe shortly. Next, we should prove that every history H in OS_C is $\mathcal{R}$-related to some history H' in OS_A. Once we finish both steps, the soundness theorem of our method lets us infer that OS_C is an observational refinement of OS_A.

The key part of the method, of course, lies in the requirement that the chosen binary relation $\mathcal{R}$ be a simulation. If we were allowed to use any relation for $\mathcal{R}$, we could pick the relation that relates all pairs of histories, and this would lead to the incorrect conclusion that every OS_C observationally refines OS_A, as long as OS_A is nonempty.

To describe our requirement on $\mathcal{R}$ and its consequence precisely, we need to formalize dependency between actions in a single trace, and define trace equivalence based on this formalization.

Definition 8 (Independent Actions). *An action φ* **is independent of** *an action φ', denoted $\varphi \# \varphi'$, iff (1) $\mathsf{getTid}(\varphi) \neq \mathsf{getTid}(\varphi')$ and (2) for all $s \in States$, $\mathsf{eval}(s, \varphi\varphi') = \mathsf{eval}(s, \varphi'\varphi)$. Here, $\mathsf{getTid}(\varphi)$ is the thread-id (i.e., the first component) of φ.*

Definition 9 (Dependency Relations). *For each trace τ, we define the* **immediate dependency relation** *$<_\tau$ to be the following relation on actions in τ:*[5] *$\tau_i <_\tau \tau_j \iff i < j \wedge \neg(\tau_i \# \tau_j)$. The* **dependency relation** *$<_\tau^+$ on τ is the transitive closure of $<_\tau$.*

Definition 10 (Trace Equivalence). *Traces τ, τ' are* **equivalent***, denoted $\tau \sim \tau'$, iff there exists a bijection $\pi : \{1, \ldots, |\tau|\} \to \{1, \ldots, |\tau'|\}$ such that $(\forall i.\ \tau_i = \tau'_{\pi(i)})$ and $(\forall i, j.\ \tau_i <_\tau^+ \tau_j \iff \tau'_{\pi(i)} <_{\tau'}^+ \tau'_{\pi(j)})$.*

Intuitively, $\tau \sim \tau'$ means that τ' can be obtained by swapping independent actions in τ. Since we swap only independent actions, we expect that τ' and τ essentially mean the same computation. The lemma below justifies this expectation, by showing that our semantics cannot observe the difference between equivalent traces.

[5] Strictly speaking, $<_\tau$ is a relation on the indices $\{1, \ldots, |\tau|\}$ of τ so that we should have written $i <_\tau j$. In this paper, we use a rather informal notation $\tau_i <_\tau \tau_j$ instead, since we found this notation easier to understand.

Lemma 2. *For all $\tau, \tau' \in WTraces$, if $\tau \sim \tau'$, then $(\forall P.\ \tau \in T(P) \iff \tau' \in T(P))$ and $(\forall s.\ \mathsf{eval}(s,\tau) = \mathsf{eval}(s,\tau'))$.*

We are now ready to give the definition of simulation, which encapsulates our requirement on relations on histories, and to prove the soundness of our proof method based on simulation.

Definition 11 (Simulation). *A binary relation $\mathcal{R}$ on histories is a* **simulation** *iff for all well-formed histories H and H' such that $(H, H') \in \mathcal{R}$,*

$$\forall \tau \in WTraces.\ \mathsf{getHistory}(\tau) = H \Longrightarrow \exists \tau' \in WTraces.\ \tau \sim \tau' \wedge \mathsf{getHistory}(\tau') = H'.$$

One way to understand this definition is to read a history H as a representation of the trace set $\mathsf{means}(H) = \{\tau \in WTraces \mid \mathsf{getHistory}(\tau) = H\}$. Intuitively, this set consists of the well-formed traces whose interactions with objects are precisely H. According to this reading, the requirement in the definition of simulation simply means that $\mathsf{means}(H)$ is a subset of $\mathsf{means}(H')$ modulo trace equivalence $\sim$. For every relation $\mathcal{R}$ on histories, we define its lifting to a relation $\lhd_{\mathcal{R}}$ on object systems as follows: $OS_C \lhd_{\mathcal{R}} OS_A \iff \forall H \in OS_C.\ \exists H' \in OS_A.\ (H, H') \in \mathcal{R}$.

Theorem 1. *If $OS_C \lhd_{\mathcal{R}} OS_A$ and $\mathcal{R}$ is a simulation, OS_C observationally refines OS_A.*

Proof. Consider a program P and states s, s' such that $s' \in [\![P]\!](OS_C)(s)$. Then, by the definition of $[\![P]\!]$, there exist a well-formed trace $\tau \in T(P)$ and a history $H \in OS_C$ such that $\mathsf{getHistory}(\tau) = H$ and $s' \in \mathsf{eval}(s,\tau)$. Since $H \in OS_C$ and $OS_C \lhd_{\mathcal{R}} OS_A$ by our assumption, there exists $H' \in OS_A$ with $(H, H') \in \mathcal{R}$. Furthermore, H and H' are well-formed, because object systems contain only well-formed histories. Now, since $\mathcal{R}$ is a simulation, τ is well-formed and $\mathsf{getHistory}(\tau) = H$, there exists a well-formed trace τ' such that (1) $\tau \sim \tau'$ and (2) $\mathsf{getHistory}(\tau') = H'$. Note that because of Lemma 2, the first conjunct here implies that $\tau' \in T(P)$ and $s' \in \mathsf{eval}(s,\tau')$. This and the second conjunct $\mathsf{getHistory}(\tau') = H'$ together imply the desired $s' \in [\![P]\!](OS_A)(s)$. □

7 Sequential Consistency, Linearizability and Refinement

Now we explain the first two main results of this paper: (1) linearizability implies observational refinement; (2) sequential consistency implies observational refinement if client operations of each thread access thread-local variables (or resources) only.

It is not difficult to obtain high-level understanding about why our results hold. Both linearizability and sequential consistency define certain relationships between two object systems, one of which is normally assumed sequential and local. Interestingly, in both cases, we can prove that these relationships are generated by lifting some *simulation* relations. From this observation follow our results, because Theorem 1 says that all such simulation-generated relationships on object systems imply observational refinements.

In the rest of this section, we will spell out the details of the high-level proof sketches just given. For this, we need to review the relations on histories used by sequential consistency and linearizability [6].

Definition 12 (Weakly Equivalent Histories). *Two histories are* **weakly equivalent***, denoted $H \equiv H'$, iff their projections to threads are equal:*[6] $H \equiv H' \iff \forall t. H|_t = H'|_t$.

As its name indicates, the weak equivalence is indeed a weak notion. It only says that the two traces are both interleavings of the same sequential threads (but they could be different interleavings).

Definition 13 (Happen-Before Order). *For each history H, the* **happen-before order** *$\prec_H$ is a binary relation on object actions in H defined by*

$$H_i \prec_H H_j \iff \exists i', j'.\ i \le i' < j' \le j \ \wedge\ \mathsf{retAct}(H_{i'}) \ \wedge\ \mathsf{callAct}(H_{j'}) \ \wedge\ \mathsf{getTid}(H_i) = \mathsf{getTid}(H_{i'}) \ \wedge\ \mathsf{getTid}(H_{j'}) = \mathsf{getTid}(H_j)$$

Here $\mathsf{retAct}(\psi)$ *holds when ψ is a return and* $\mathsf{callAct}(\psi)$ *holds when ψ is a call.*

This definition is intended to express that in the history H, the method call for H_i is completed before the call for H_j starts. To see this intention, assume that H is well-formed. One important consequence of this assumption is that if an object action ψ of some thread t is followed by some return action ψ' of the same thread in the history H (i.e., $H = ...\psi...\psi'...$), then the return for ψ itself appears before ψ' or it is ψ'. Thus, the existence of $H_{i'}$ in the definition ensures that the return action for H_i appears before or at $H_{i'}$ in the history H. By a similar argument, we can see that the call for H_j appears after or at $H_{j'}$. Since $i' < j'$, these two observations mean that the return for H_i appears before the call for H_j, which is the intended meaning of the definition. Using this happen-before order, we define the linearizability relation $\sqsubseteq$:

Definition 14 (Linearizability Relation). *The linearizability relation is a binary relation $\sqsubseteq$ on histories defined as follows: $H \sqsubseteq H'$ iff (1) $H \equiv H'$ and (2) there is a bijection $\pi : \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that*[7] $(\forall i.\, H_i = H'_{\pi(i)})$ *and* $(\forall i, j.\, H_i \prec_H H_j \Longrightarrow H'_{\pi(i)} \prec_{H'} H'_{\pi(j)})$.

Recall that for each relation $\mathcal{R}$ on histories, its lifting $\lhd_{\mathcal{R}}$ to the relation on object systems is defined by: $OS \lhd_{\mathcal{R}} OS' \iff \forall H \in OS.\, \exists H' \in OS'.\, (H, H') \in \mathcal{R}$. Using this lifting, we formally specify sequential consistency and linearizability.

Definition 15. *Let OS_A and OS_C be object systems. We say that OS_C is* **sequentially consistent** *wrt. OS_A iff $OS_C \lhd_{\equiv} OS_A$. We also say that OS_C is* **linearizable** *wrt. OS_A iff $OS_C \lhd_{\sqsubseteq} OS_A$.*

Note that this definition does not assume the sequentiality and locality of OS_A, unlike Herlihy and Wing's definitions. We use this more general definition here in order to emphasize that the core ideas of sequential consistency and linearizability lie in relations $\equiv$ and $\sqsubseteq$ on histories, not in the use of a sequential local object system (as a specification).

We first prove the theorem that connects linearizability and observational refinement. Our proof uses the lemma below:

[6] For the same definition, Herlihy and Wing use the terminology "equivalence".

[7] In this paper, we consider only those histories that arise from complete terminating computations; see the definition of $[\![P]\!]$ in Section 5. Consequently, we do not have to worry about completing or removing pending calls in histories, unlike Herlihy and Wing's definition.

Lemma 3. *Let H be a well-formed history and let i, j be indices in $\{1, \ldots, |H|\}$. Then,*

$$(\exists \tau \in \mathit{WTraces}.\ \mathsf{getHistory}(\tau) = H \ \wedge \ H_i <_\tau^+ H_j) \\ \Longrightarrow \ (i < j) \ \wedge \ (\mathsf{getTid}(H_i) = \mathsf{getTid}(H_j) \ \vee \ H_i \prec_H H_j).$$

Proof. Consider a well-formed history H, indices i, j of H and a well-formed trace τ such that the assumptions of this lemma hold. Then, we have indices $i_1 < i_2 < \ldots < i_n$ of τ such that

$$H_i \ = \ \tau_{i_1} \ <_\tau \ \tau_{i_2} \ <_\tau \ \ldots \ <_\tau \ \tau_{i_{n-1}} \ <_\tau \ \tau_{i_n} \ = \ H_j. \quad (1)$$

One conclusion $i < j$ of this lemma follows from this, because $\mathsf{getHistory}(\tau) = H$ means that the order of object actions in H are maintained in τ. To obtain the other conclusion of the lemma, let $t = \mathsf{getTid}(H_i)$ and $t' = \mathsf{getTid}(H_j)$. Suppose that $t \neq t'$. We will prove that for some $i_k, i_l \in \{i_1, \ldots, i_n\}$,

$$i_k < i_l \ \wedge \ t = \mathsf{getTid}(\tau_{i_k}) \ \wedge \ t' = \mathsf{getTid}(\tau_{i_l}) \ \wedge \ \mathsf{retAct}(\tau_{i_k}) \ \wedge \ \mathsf{callAct}(\tau_{i_l}). \quad (2)$$

Note that this gives the conclusion we are looking for, because all object actions in τ are from H and their relative positions in τ are the same as those in H. In the rest of the proof, we focus on showing (2) for some i_k, i_l. By the definition of $\#$, an object action ψ can depend on another action φ, only when both actions are done by the same thread. Now note that the first and last actions in the chain in (1) are *object* actions by *different* threads t and t'. Thus, the chain in (1) must contain *client* operations τ_{i_x} and τ_{i_y} such that $\mathsf{getTid}(\tau_{i_x}) = t$ and $\mathsf{getTid}(\tau_{i_y}) = t'$. Let τ_{i_a} be the first client operation by the thread t in the chain and let τ_{i_b} be the last client operation by t'. Then, $i_a < i_b$. This is because otherwise, the sequence $\tau_{i_a} \ \tau_{i_a+1} \ \ldots \ \tau_{i_n}$ does not have any client operation of the thread t', while τ_{i_a} is an action of the thread t and τ_{i_n} is an action of the different thread t'; these facts make it impossible to have $\tau_{i_a} <_\tau \tau_{i_a+1} <_\tau \ldots <_\tau \tau_{i_n}$. Since τ_{i_1} is an object action by the thread t and τ_{i_a} is a client operation by the same thread, by the well-formedness of τ, there should exist some i_k between i_1 (including) and i_a such that τ_{i_k} is a return object action by the thread t. By a symmetric argument, there should be some i_l between i_b and i_n (including) such that τ_{i_l} is a call object action by t'. We have just shown that i_k and i_l satisfy (2), as desired. □

Theorem 2. *The linearizability relation $\sqsubseteq$ is a simulation.*

Proof. For an action φ and a trace τ, define $\varphi \# \tau$ to mean that $\varphi \# \tau_j$ for all $j \in \{1, \ldots, |\tau|\}$. In this proof, we will use this $\varphi \# \tau$ predicate and the following facts:

Fact 1. Trace equivalence $\sim$ is symmetric and transitive.
Fact 2. If $\tau \sim \tau'$ and τ is well-formed, τ' is also well-formed.
Fact 3. If $\tau\tau'$ is well-formed, its prefix τ is also well-formed.
Fact 4. If $\varphi \# \tau'$, we have that $\tau\varphi\tau' \sim \tau\tau'\varphi$.
Fact 5. If $\tau \sim \tau'$, we have that $\tau\varphi \sim \tau'\varphi$.

Consider well-formed histories H, S and a well-formed trace τ such that $H \sqsubseteq S$ and $\mathsf{getHistory}(\tau) = H$. We will prove the existence of a trace σ such that $\tau \sim \sigma$ and

$\mathsf{getHistory}(\sigma) = S$. This gives the desired conclusion of this theorem; the only missing requirement for proving that $\sqsubseteq$ is a simulation is the well-formedness of σ, but it can be inferred from $\tau \sim \sigma$ and the well-formedness of τ by Fact 2.

Our proof is by induction on the length of S. If $|S| = 0$, H has to the empty sequence as well. Thus, we can choose τ as the required σ in this case. Now suppose that $|S| \neq 0$. That is, $S = S'\psi$ for some history S' and object action ψ. Note that since the well-formed traces are closed under prefix (Fact 3), S' is also a well-formed history. During the proof, we will use this fact, especially when applying induction on S'.

Let δ be the projection of τ to client operations (i.e., $\delta = \tau|_{\mathcal{A}_c}$). The starting point of our proof is to split τ, H, δ. By assumption, $H \sqsubseteq S'\psi$. By the definition of $\sqsubseteq$, this means that

$$\begin{array}{l}\exists H', H''.\ H = H'\psi H'' \ \wedge\ H'H'' \sqsubseteq S' \\ \qquad \wedge\ \big(\forall j \in \{1, \ldots, |H''|\}.\ \neg(\psi \prec_H H''_j) \ \wedge\ \mathsf{getTid}(\psi) \neq \mathsf{getTid}(H''_j)\big).\end{array} \tag{3}$$

Here we use the bijection between indices of H and $S'\psi$, which exists by the definition of $H \sqsubseteq S'\psi$. The action ψ in $H'\psi H''$ is what is mapped to the last action in $S'\psi$ by this bijection. The last conjunct of (3) says that the thread-id of every action of H'' is different from $\mathsf{getTid}(\psi)$. Thus, $\psi \# H''$ (because an *object* action is independent of all actions by *different* threads). From this independence and the well-formedness of H, we can drive that $H'H''\psi$ is well-formed (Facts 2 and 4), and that its prefix $H'H''$ is also well-formed (Fact 3). Another important consequence of (3) is that since $\tau \in interleave(\delta, H)$, the splitting $H'\psi H''$ of H induces splittings of τ and δ as follows: there exist $\tau', \tau'', \delta', \delta''$ such that

$$\tau = \tau'\psi\tau'' \wedge \delta = \delta'\delta'' \wedge \tau' \in interleave(\delta', H') \wedge \tau'' \in interleave(\delta'', H''). \tag{4}$$

The next step of our proof is to identify one short-cut for showing this theorem. The short-cut is to prove $\psi \# \tau''$. To see why this short-cut is sound, suppose that $\psi \# \tau''$. Then, by Fact 4,

$$\tau \ = \ \tau'\psi\tau'' \ \sim \ \tau'\tau''\psi. \tag{5}$$

Since τ is well-formed, this implies that $\tau'\tau''\psi$ and its prefix $\tau'\tau''$ are well-formed traces as well (Facts 2 and 3). Furthermore, $\mathsf{getHistory}(\tau'\tau'') = H'H''$, because of the last two conjuncts of (4). Thus, we can apply the induction hypothesis to $\tau'\tau'', H'H'', S'$, and obtain σ with the property: $\tau'\tau'' \sim \sigma \ \wedge\ \mathsf{getHistory}(\sigma) = S'$. From this and Fact 5, it follows that

$$\tau'\tau''\psi \sim \sigma\psi \quad \wedge \quad \mathsf{getHistory}(\sigma\psi) = \mathsf{getHistory}(\sigma)\psi = S'\psi. \tag{6}$$

Now, the formulas (5) and (6) and the transitivity of $\sim$ (Fact 1) imply that $\sigma\psi$ is the required trace by this theorem. In the remainder of the proof, we will use this short-cut, without explicitly mentioning it.

The final step is to do the case analysis on δ''. Specifically, we use the nested induction on the length of δ''. Suppose that $|\delta''| = 0$. Then, $\tau'' = H''$, and by the last conjunct of (3) (the universal formula), we have that $\psi \# \tau''$; since ψ is an object action, it is independent of actions by different threads. The theorem follows from this. Now consider the inductive case of this nested induction: $|\delta''| > 0$. Note that if $\psi \# \delta''$, then

$\psi\#\tau''$, which implies the theorem. So, we are going to assume that $\neg(\psi\#\delta'')$. Pick the greatest index i of τ'' such that $\psi <_\tau^+ \tau_i''$. Let $\varphi = \tau_i''$. Because of the last conjunct of (3) and Lemma 3, τ_i'' comes from δ, not H''. In particular, this ensures that there are following further splittings of δ'', τ'' and H'': for some traces $\gamma, \gamma', \kappa, \kappa', T, T'$,

$$\begin{array}{l}\delta'' \;=\; \gamma\varphi\gamma' \;\wedge\; \tau'' \;=\; \kappa\varphi\kappa' \;\wedge\; H'' \;=\; TT' \;\wedge \\ \kappa \;\in\; \mathit{interleave}(\gamma, T) \;\wedge\; \kappa' \;\in\; \mathit{interleave}(\gamma', T') \;\wedge\; \varphi\#\kappa'.\end{array}$$

Here the last conjunct $\varphi\#\kappa'$ comes from the fact that φ is the last element of τ'' with $\psi <_\tau^+ \varphi$. Since γ' is a subsequence of κ', the last conjunct $\varphi\#\kappa'$ implies that $\varphi\#\gamma'$. Also, $\tau'\psi\kappa\varphi\kappa' \sim \tau'\psi\kappa\kappa'\varphi$ by Fact 4. Now, since $\tau = \tau'\psi\kappa\varphi\kappa'$ is well-formed, the equivalent trace $\tau'\psi\kappa\kappa'\varphi$ and its prefix $\tau'\psi\kappa\kappa'$ both are well-formed as well (Facts 2 and 3). Furthermore, $\tau'\psi\kappa\kappa' \in \mathit{interleave}(\delta'\gamma\gamma', H'\psi H'')$. Since the length of $\gamma\gamma'$ is shorter than δ'', we can apply the induction hypothesis of the nested induction, and get

$$\exists\sigma. \quad \tau'\psi\kappa\kappa' \sim \sigma \quad\wedge\quad \mathsf{getHistory}(\sigma) = S'\psi. \tag{7}$$

We will prove that $\sigma\varphi$ is the trace desired for this theorem. Because of $\varphi\#\kappa'$ and Fact 4, $\tau = \tau'\psi\kappa\varphi\kappa' \sim \tau'\psi\kappa\kappa'\varphi$. Also, because of Fact 5 and the first conjunct of (7), $\tau'\psi\kappa\kappa'\varphi \;\sim\; \sigma\varphi$. Thus, $\tau \sim \sigma\varphi$ by the transitivity of $\sim$. Furthermore, since φ is not an object action, the second conjunct of (7) implies that $\mathsf{getHistory}(\sigma\varphi) = \mathsf{getHistory}(\sigma) = S'\psi$. We have just shown that $\sigma\varphi$ is the desired trace. □

Corollary 1. *If OS_C is linearizable wrt. OS_A, then OS_C observationally refines OS_A.*

Next, we consider sequential consistency. For sequential consistency to imply observational refinement, we need to restrict programs such that threads can access local variables only in their client operations: $\forall t, t', a, a'.\, (t{\neq}t' \wedge a \in \mathit{Cop}_t \wedge a' \in \mathit{Cop}_{t'}) \Longrightarrow a \,\#\, a'$.

Lemma 4. *Suppose that all threads can access local variables only in their client operations. Then, for all well-formed histories H and indices i, j in $\{1, \ldots, |H|\}$,*

$$(\exists\tau{\in}\mathit{WTraces}.\, \mathsf{getHistory}(\tau){=}H \wedge H_i{<}_\tau^+ H_j) \Longrightarrow i{<}j \wedge \mathsf{getTid}(H_i){=}\mathsf{getTid}(H_j).$$

Proof. Consider a well-formed history H, indices i, j and a well-formed trace τ satisfying the assumptions of this lemma. Then, for some indices $i_1 < \ldots < i_n$ of τ,

$$H_i \;=\; \tau_{i_1} \;<_\tau\; \tau_{i_2} \;<_\tau\; \ldots \;<_\tau\; \tau_{i_{n-1}} \;<_\tau\; \tau_{i_n} \;=\; H_j. \tag{8}$$

One conclusion $i < j$ of this lemma follows from this; the assumption $\mathsf{getHistory}(\tau) = H$ of this lemma means that the order of object actions in H are maintained in τ. To obtain the other conclusion of the lemma, we point out one important property of $\#$: under the assumption of this lemma, $\neg(\varphi\#\varphi')$ only when $\mathsf{getTid}(\varphi) = \mathsf{getTid}(\varphi')$. (Here φ, φ' are not necessarily object actions.) To see why this property holds, we assume $\neg(\varphi\#\varphi')$ and consider all possible cases of φ and φ'. If one of φ and φ' is an object action, the definition of $\#$ implies that φ and φ' have to be actions by the same thread. Otherwise, both φ and φ' are atomic client operations. By our assumption, all threads

access only local variables in their client operations, so that two client operations are independent if they are performed by different threads. This implies that φ and φ' should be actions by the same thread. Now, note that $\tau_k <_\tau \tau_l$ implies $\neg(\tau_k \# \tau_l)$, which in turn entails $\mathsf{getTid}(\tau_k) = \mathsf{getTid}(\tau_l)$ by what we have just shown. Thus, we can derive the following desired equality from (8): $\mathsf{getTid}(H_i) = \mathsf{getTid}(\tau_{i_1}) = \mathsf{getTid}(\tau_{i_2}) = \ldots = \mathsf{getTid}(\tau_{i_n}) = \mathsf{getTid}(H_j)$. □

Theorem 3. *If all threads access local variables only in their client actions, the weak equivalence $\equiv$ is a simulation.*

Proof. The proof is similar to the one for Theorem 2. Instead of repeating the common parts between these two proofs, we will explain what we need to change in the proof of Theorem 2, so as to obtain the proof of this theorem. Firstly, we should replace linearizability relation $\sqsubseteq$ by weak equivalence $\equiv$. Secondly, we need to change the formula (3) to

$$\exists H'H''.\ H = H'\psi H'' \wedge H'H'' \equiv S' \wedge \forall j \in \{1, ..., |H''|\}.\ \mathsf{getTid}(\psi) \neq \mathsf{getTid}(H''_j).$$

Finally, we should use Lemma 4 instead of Lemma 3. After these three changes have been made, the result becomes the proof of this theorem. □

Corollary 2. *If OS_C is sequentially consistent wrt. OS_A and all threads access local variables only in their client actions, OS_C is an observational refinement of OS_A.*

Completeness. Under suitable assumptions on programming languages and object systems, we can obtain the converse of Corollaries 1 and 2: observational refinement implies linearizability and sequential consistency. First, we assume that object systems OS contain only those histories all of whose calls have matching returns. This assumption is necessary, because observational refinement considers only terminating, completed computations. Next, we assume that threads' primitive commands include the `skip` statement. Finally, we consider specific assumptions for sequential consistency and linearizability, which will be described shortly.

For sequential consistency, we suppose that the programming language contains atomic assignments $x{:=}n$ of constants n to thread-local variable x and has atomic assume statements of the form $\texttt{assume}(x{=}n)$ with thread-local variable x.[8] Note that this supposition does not require the use of any global variables, so that it is consistent with the assumption of Corollary 2. Under this supposition, observational refinement implies sequential consistency.

Theorem 4. *If OS_C observationally refines OS_A then $OS_C \lhd_\equiv OS_A$.*

The main idea of the proof is to create for every history $H \in OS_C$ a program P_H that records the interaction of every thread t with the object system using t's local variables. For the details of the proof, see the full version of this paper [3].

For linearizability, we further suppose that there is a single global variable g shared by all threads. That is, threads can assign constants to g atomically, or they can run

[8] Technically, this assumption also means that $T(x{:=}n)t$ and $T(\texttt{assume}(x{=}n))t$ are singleton traces (t, a) and (t, b), where $[\![b]\!](s) \equiv \mathsf{if}\ (s(x){=}n)\ \mathsf{then}\ \{s\}\ \mathsf{else}\ \{\}$ and $[\![a]\!](s) \equiv \{s[x{\mapsto}n]\}$.

the statement $\texttt{assume}(g{=}n)$ for some constant n. Under this supposition, observational refinement implies linearizability.

Theorem 5. *If OS_C observationally refines OS_A, then $OS_C \lhd_{\sqsubseteq} OS_A$.*

The core idea of the proof is, again, to create for every history $H \in OS_C$ one specific program P_H. This program uses a single global variable and satisfies that for every (terminating) execution τ of P_H, the object history of τ always has the same happen-before relation as H. See the full paper [3] for the details.

8 Abstract Dependency

Although our results on observational refinements give complete characterization of sequential consistency and linearizability, they still do not explain where the relations $\equiv$ and $\sqsubseteq$ in sequential consistency and linearizability come from. In this section, we will answer this question using the dependency between actions.

The result of this section is based on one reading of a well-formed history H. In this reading, the history H means not the single trace H itself but the set of all the well-formed traces whose object actions are described by H. Formally, we let $WHist$ be the set of all the well-formed histories, and define function $\mathsf{means} : WHist \to \mathcal{P}(WTraces)$ by $\mathsf{means}(H) = \{\tau \in WTraces \mid \mathsf{getHistory}(\tau) = H\}$.

Using means, we define a new relation on well-formed histories, which compare possible dependencies between actions in the histories.

Definition 16 (Abstract Dependency). *For each well-formed history H, the* **abstract dependency** $<_H^{\#}$ *for H is the binary relation on actions in H determined as follows: $H_i <_H^{\#} H_j \iff i < j \wedge \exists \tau \in \mathsf{means}(H).\ H_i <_\tau^{+} H_j$.*

Definition 17 (Causal Complexity Relation). *The* **causal complexity relation** $\sqsubseteq^{\#}$ *is a binary relation on well-formed histories, such that $H \sqsubseteq^{\#} S$ iff there exists a bijection $\pi : \{1, \ldots, |H|\} \to \{1, \ldots, |S|\}$ satisfying (1) $\forall i \in \{1, \ldots, |H|\}.\ H_i = S_{\pi(i)}$ and (2) $\forall i, j \in \{1, \ldots, |H|\}.\ H_i <_H^{\#} H_j \Longrightarrow S_{\pi(i)} <_S^{\#} S_{\pi(j)}$.*

Intuitively, $H \sqsubseteq^{\#} S$ means that S is a rearrangement of actions in H that preserves all the abstract causal dependencies in H. Note that S might contain abstract causal dependencies that are not present in H.

The result below shows when sequential consistency or linearizability coincides with causal complexity relation.

Theorem 6. *If all threads access only local variables in their client operations, then $\forall H, S \in WHist.\ H \equiv S \iff H \sqsubseteq^{\#} S$.*

Theorem 7. *Assume that for every pair (t, t') of thread-ids with $t \neq t'$, there exist client operations $a \in Cop_t$ and $a' \in Cop_{t'}$ with $\neg(a \# a')$. Under this assumption, we have the following equivalence: $\forall H, S \in WHist.\ H \sqsubseteq S \iff H \sqsubseteq^{\#} S$.*

9 Conclusions

Developing a theory of data abstraction in the presence of concurrency has been a long-standing open question in the programming language community. In this paper, we have shown that this open question can be attacked from a new perspective, by carefully studying correctness conditions proposed by the concurrent-algorithm community, using the tools of programming languages. We prove that linearizability is a sound method for proving observational refinements for concurrent objects, which is complete when threads are allowed to access shared global variables. When threads access only thread-local variables, we have shown that sequential consistency becomes a sound and complete proof method for observational refinements. We hope that our new understanding on concurrent objects can facilitate the long-delayed transfer of the rich existing theories of data-abstraction [7,8,13,10,12] from sequential programs to concurrent ones.

In the paper, we used a standard assumption on a programming language from the concurrent-algorithm community. We assumed that a programming language did not allow callbacks from concurrent objects to client programs, that all the concurrent objects were properly encapsulated [1], and that programs were running under "sequentially consistent" memory models. Although widely used by the concurrent-algorithm experts, these assumptions limit the applicability of our results. In fact, they also limit the use of linearizability in the design of concurrent data structures. Removing these assumptions and extending our results is what we plan to do next.

References

1. Banerjee, A., Naumann, D.A.: Representation independence, confinement and access control. In: POPL 2002 (2002)
2. Brookes, S.D.: A semantics for concurrent separation logic. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 16–34. Springer, Heidelberg (2004)
3. Filipović, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Technical report, Queen Mary University of London (December 2008)
4. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213. Springer, Heidelberg (1986)
5. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
6. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
7. Hoare, C.A.R.: Proof of correctness of data representations. Acta Inf. 1, 271–281 (1972)
8. Hoare, C.A.R., He, J., Sanders, J.W.: Prespecification in data refinement. Inf. Proc. Letter 25(2), 71–76 (1987)
9. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
10. Mitchell, J., Plotkin, G.: Abstract types have existential types. ACM TOPLAS 10(3), 470–502 (1988)
11. Plotkin, G.: LCF considered as a programming language. TCS 5, 223–255 (1977)
12. Plotkin, G., Abadi, M.: A logic for parametric polymorphism. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664. Springer, Heidelberg (1993)
13. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) Information Processing 1983, pp. 513–523. North-Holland, Amsterdam (1983)