M.S. THESIS

OPERATIONAL SEMANTICS FOR EXPRESSING AND REASONING ABOUT FAIRNESS PROPERTIES

프로그램의 공정성 성질들을 표현하고 이용하기 위한 실행 의미에 관한 연구

February 2024

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Dongjae Lee

OPERATIONAL SEMANTICS FOR EXPRESSING AND REASONING ABOUT FAIRNESS PROPERTIES

프로그램의 공정성 성질들을 표현하고 이용하기 위한 실행 의미에 관한 연구

지도교수 허충길

이 논문을 공학석사 학위논문으로 제출함

2024년 01월

서울대학교 대학원

컴퓨터공학부

이 동 재

이동재의 공학석사 학위논문을 인준함

2024년 01월

Abstract

This thesis presents FOS (Fair Operational Semantics), a theory capable of expressing various notions of fairness in the form of operational semantics and enabling reasoning principles for these notions of fairness. Fairness properties state that a sequence of bad events cannot happen infinitely before a good event takes place. These properties are often crucial in program verification in two ways. First, fairness properties can serve as a specification, enabling a more precise description of an implementation beyond safety. Second, program verification, such as the verification of termination of a concurrent program, often depends on fairness properties. This work shows that FOS is useful in both cases by developing specifications that represent various concepts of fairness properties, developing thread-local verification techniques for fairness properties, and verifying examples based on these techniques.

Keywords: Operational Semantics, Concurrency, Fairness, Program Verification

Student Number: 2021-26995

Contents

List of Figures

Chapter 1

Introduction

In program verification, fairness properties, which state that a sequence of bad events cannot happen infinitely before a good event takes place, are often crucial in two ways. First, fairness properties can serve as a part of a specification, enabling a more precise description of an implementation beyond safety. Second, verification of a program, such as termination, can depend on fairness properties. However, general methods for expressing and reasoning about various kinds of fairness properties are relatively underdeveloped compared to those for safety properties.

This thesis presents FOS (Fair Operational Semantics), a theory capable of expressing various notions of fairness in the form of operational semantics and enabling reasoning principles for these notions of fairness. This work shows that FOS is useful by developing specifications that represent various concepts of fairness properties, developing thread-local verification techniques for fairness properties, and verifying examples based on these techniques. Specifically, FOS enables thread-local reasoning about fairness by providing thread-local simulation relations equipped with separation-logic-style resource algebras. This technique is used in the verification of a ticket lock implementation and a client of the ticket lock under weak memory concurrency, which requires reasoning about different notions of fairness including the fairness of a scheduler, fairness of the ticket lock implementation, and even fairness of weak memory.

The result of this thesis was published in PLDI'2023 under the title “Fair Operational Semantics” [18]. Also, the theory of FOS and the examples are formalized in the Coq proof assistant.

Chapter 2

Fair Operational Semantics

2.1 Introduction: Fair Operational Semantics

Although safety properties (*i.e.,* something bad never happens under a certain condition) have been major goals of verification, fairness properties (*i.e.,* something bad cannot happen indefinitely without anything good occurring) are also often crucial in verification, in particular, for concurrent programs. For example, fairness assumptions about the underlying system—such as fairness about the scheduler (*i.e.,* a thread cannot be delayed by the scheduler indefi-

```
               ‖ do {                 ‖
lock_abs();    ‖   lock_abs();        ‖
X := 42;       ‖   x := X;            ‖
unlock_abs();  ‖   unlock_abs();   ⊑    skip;  ‖ print(42);   (CL_S)
               ‖ } while (x = 0)      ‖
               ‖ print(x);            ‖
                      (CL_I)
```

Figure 2.1 An example of concurrent program verification.

nitely), or fairness about weak memory (*i.e.,* memory cannot delay committing a write indefinitely)—are often required to verify concurrent programs. Moreover, customized concepts of fairness are required when working with custom libraries: for example, fairness about acquiring a lock provided by a custom ticket lock library (*i.e.,* a lock request cannot fail indefinitely given that the lock is available).

However, general methods for abstractly expressing various concepts of fairness and reasoning about them are relatively underdeveloped. For example, separation logics such as Iris [12] provide a flexible and powerful mechanism for specifying and reasoning about arbitrary safety properties. In contrast, existing work in the fairness domain such as TaDA-Live [5] and LiLi [22] are comparatively limited in their power, foremost in that they can only handle fixed notions of fairness instead of providing a general mechanism for expressing and reasoning about custom fairness properties.

Clearly, it is desirable to have a general mechanism capable of (*i*) capturing various kinds of fairness, and (*ii*) providing reasoning principles to both validate that a certain implementation is fair and verify client code assuming fairness of components. Consider Fig. 2.1 as an example, where a client program $\mathrm{CL_I}$ consisting of two threads utilizes a library-provided lock to protect a shared location X initialized to 0. The first thread sets X to 42, while the second thread repeatedly reades from X until reading a non-zero value, upon which it will print the read value.

Intuitively, $\mathrm{CL_I}$ *refines* the specification $\mathrm{CL_S}$ (where the first thread does nothing and the second prints 42) *assuming* that the scheduler and lock are 'fair'. For example, $\mathrm{CL_I}$ fails to refine $\mathrm{CL_S}$ if the scheduler is unfair and schedules thread 2 only; even if the scheduler is fair, an unfair lock giving the lock only to thread 2, would result in an execution trace of $\mathrm{CL_I}$ that cannot be captured by $\mathrm{CL_S}$.

Thus to prove that $\mathrm{CL_I}$ refines $\mathrm{CL_S}$, one requires (*i*) a flexible mechanism for *expressing* the fairness of schedulers and locks, and (*ii*) *reasoning principles* for reasoning about such flexible fairness properties. Moreover, the ability to express and reason about general fairness properties enables a proof to be split in two ways: (*i*) one may *exploit* the fairness assumptions when verifying client code (*e.g.,* proving that $\mathrm{CL_I}$ refines $\mathrm{CL_S}$), and (*ii*) separately *validate* that a specific implementation satisfies the fairness assumption (*e.g.,* proving that a specific scheduler or a lock is actually fair).

FOS, a Theory for Fairness. In this paper, we present **FOS** (Fair Operational Semantics), a theory for *expressing and reasoning about fairness* as an operational semantics. FOS provides:

- The standard notion of operational semantics extended with a special kind of events, called *fairness events*, that allows expressing arbitrary kinds of custom fairness properties.
- A simple global simulation relation that allows one to validate (for libraries) and exploit (for clients) fairness when proving refinements.
- A thread-local version of the aforementioned simulation and a logic capable of employing separation-logic style reasoning to perform thread-local reasoning and simplify proofs.

We demonstrate the generality and power of FOS by (*i*) specifying fairness of the scheduler, locks, and weak memory; (*ii*) proving fairness of the scheduler for a simple scheduler implementation and fairness of the lock for a ticket lock implementation; and (*iii*) exploiting these three kinds of fairness properties to verify client code (*e.g.,* $\mathrm{CL_I} \sqsubseteq \mathrm{CL_S}$). Note that existing work for reasoning about fairness is based on simple sequentially consistent concurrency, whereas we use FOS to model a (fair) weak memory concurrency model and verify our examples under the model. The theory of FOS is mechanized in the Coq proof assistant [30].

Paper Structure. The remainder of the paper is structured as follows. §2.2 first illustrates how fairness properties can be expressed and reasoned about in a global fashion in FOS. §2.3 extends these ideas to show how FOS expresses fairness under concurrency, and shows how thread-local reasoning about fairness can be achieved in FOS. §2.4 and §2.5 formalize the high-level ideas presented in §2.2 and §2.3. §2.6 presents the module system of FOS, used to modularize proofs and define specifications capturing various notions of fairness, and §2.7 presents the details of the aforementioned thread-local reasoning. §2.8 presents a high-level overview of the verification of our motivating example. §2.9 concludes with related work.

2.2 Main Ideas: Formally Expressing Fairness

In this section, we aim to illustrate the key abilities of FOS: (*i*) How can we abstractly encode assumptions about fairness? (§2.2.1), (*ii*) How can one prove that a client program refines its specification assuming fairness about a library? (§2.2.2), and (*iii*) How can one prove that a library implementation refines its specification, validating the fairness assumption about the library? (§2.2.3).

We illustrate these abilities via the following example LOT, which draws a Boolean value via a lottery function `lottery`(), and prints "win!" whenever that value is *true*:

$$\texttt{loop}\ \{\ \texttt{if lottery}()\ \texttt{then}\ \mathit{print}(\text{"win!"})\ \texttt{else skip}\ \} \tag{LOT}$$

Note that LOT is *sequential* even though fairness commonly comes into play when reasoning about concurrent programs. Nevertheless, LOT is expressive enough to illustrate the key ideas behind FOS. The fairness assumption in LOT is that `lottery`() cannot return *false* indefinitely without returning *true*. Under this assumption, LOT prints infinitely many "win!"s, thus *refining* the following program:

$$\texttt{loop}\ \{\ \mathit{print}(\text{"win!"})\ \} \tag{WIN}$$

The goal of this section is to first show how the assumption that `lottery`() is fair can be encoded as a semantics of code, which doubles as a specification (§2.2.1), prove that LOT indeed refines WIN under this assumption (§2.2.2), and finally show how a concrete implementation of `lottery`() can be shown to validate this assumption (§2.2.3).

2.2.1 Fairness as a Semantics

As briefly mentioned in §2.1, fairness is the concept of ensuring that only a finite number of 'bad events' happen before a 'good event' takes place. FOS captures this concept by defining two *fairness events*: `good` and `bad`. Intuitively, `good` is triggered when a good event happens (*e.g.,* `lottery`() triggers `good` when returning *true*), and `bad` is triggered when a bad event happens (*e.g.,* `lottery`() triggers `bad` when returning *false*).

Observe that in a general setting, there are typically many *entities* for which fairness should be ensured (*e.g.,* each thread in a scheduler). In such a setting, a specific event (*e.g.,* thread 1 being scheduled) may be `good` for certain entities (*e.g.,* thread 1) and `bad` for other entities (*e.g.,* the other threads). FOS formalizes this intuition by triggering fairness events according to a f**airness map** `fmap`, which maps a *fairness id* to a fairness event `good` or `bad`. A fairness id is a unique identifier for each entity vying for fairness, *one for each kind of fairness* one wishes to consider: for example, each thread would have two fairness ids in a scenario considering both lock and scheduler fairness.

In this example, there is only one entity that needs fairness to be ensured (the program), and only one kind of fairness (the lottery), hence there is only one fairness id lot. Thus our `fmap` is of type $\texttt{fmap} : \{\mathsf{lot}\} \mapsto \{\texttt{good}, \texttt{bad}\}$.[1]

In order to actually trigger the fairness events within a fairness map, one requires a construct to trigger these events in code. This role is fulfilled by the

[1]A formal definition of `fmap` includes ϵ (an empty event) in the codomain, but we choose to ignore this for the time being.

fairness constructor `FAIR` in FOS, whose basic semantics is to take as argument an `fmap` and trigger all fairness events within the map for the appropriate fairness ids.

Modelling fairness through `good` and `bad` events that are triggered explicitly throughout the program allows us to give a formal definition of what it means for a program trace to be fair:

Definition 2.2.1 (Fair Trace) *Consider a program P equipped with a set of fairness ids* $\mathtt{ID}$*, and a trace t obtained by executing P. For a fairness id $\mathtt{id} \in \mathtt{ID}$, let $t_{\mathtt{id}}$ be a sub-sequence of events of t obtained by taking only the fairness events that are associated with* $\mathtt{id}$*.*

We say that t is a fair trace *of P iff every $t_{\mathtt{id}}$ does not contain an infinite sequence of `bad` events that precedes a `good` event. Otherwise, we say that t is an* unfair trace *of P. If all possible traces of P are fair traces, then we say that P* is fair*.*

Following Definition 2.2.1, consider the following specification of `lottery()` that encodes fairness of `lottery()` for lot ($\mathtt{PICK}(\mathbb{B})$ nondeterministically picks a Boolean value):

$$\begin{aligned}\mathtt{def\ lottery}()_{\mathtt{spec}} = \mathtt{if\ PICK}(\mathbb{B})\ \mathtt{then\ FAIR}([\mathsf{lot} \mapsto \mathtt{good}]);\ \mathtt{ret}\ \mathit{true} \\ \mathtt{else\ FAIR}([\mathsf{lot} \mapsto \mathtt{bad}]);\ \mathtt{ret}\ \mathit{false}\end{aligned} \qquad \text{(SPEC)}$$

Here `FAIR` takes the fmap $[\mathsf{lot} \mapsto \mathtt{good}]$ when `lottery()` returns *true*, which captures that a good event for lot happens. On the other hand, `FAIR` takes $[\mathsf{lot} \mapsto \mathtt{bad}]$ on the false branch, capturing that a bad event for lot has occurred.

As stated in Definition 2.2.1, a fair execution (trace) of SPEC is a trace where each fairness id does not accumulate an infinite number of `bad` events before a `good` event. In tandem with triggering fairness events, the fairness constructor `FAIR` also *filters out* such unfair traces. A good way to understand `FAIR` is as a monitor that keeps track of the fairness events that each fairness id sees via the supplied fairness maps, and filters unfair executions out. One can then understand that the following unfair execution trace is *not* a valid trace of SPEC, as `FAIR` *prohibits* such unfair traces:

$$[\mathsf{lot} \mapsto \texttt{bad}] :: [\mathsf{lot} \mapsto \texttt{bad}] :: [\mathsf{lot} \mapsto \texttt{bad}] :: [\mathsf{lot} \mapsto \texttt{bad}] :: \ldots$$

On the other hand, traces such as the following are allowed, as each `bad` has a following `good`:

$$[\mathsf{lot} \mapsto \texttt{bad}] :: [\mathsf{lot} \mapsto \texttt{good}] :: [\mathsf{lot} \mapsto \texttt{bad}] :: [\mathsf{lot} \mapsto \texttt{good}] :: \ldots$$

In the end, the fairness events `good` and `bad` are silent events that have no effect on the actual behavior of the program; they are merely used to encode fairness via `FAIR`. Thus SPEC becomes a valid fair specification for `lottery`() that captures exactly the 'fair' behavior of a lottery.

As shown, FOS allows one to encode desired notions of fairness *directly as a piece of code but still in an abstract form* through the use of fairness maps and `FAIR`. This puts us in a very favorable position for our next task, to prove that LOT refines WIN assuming that `lottery`() is fair, as one may now utilize SPEC directly to build the refinement proof.

2.2.2 Proving that LOT Refines WIN: Exploiting Fairness

Based on our encoding of fairness as a semantics, we now wish to prove that LOT where the semantics of `lottery`() are given by SPEC (which we denote as LOT[SPEC]) refines WIN. In proving refinement, we take the standard approach of constructing a simulation that matches arbitrary program steps from the target to program steps of the source.

What makes refinement in FOS special is the semantics of `FAIR`, which filters out any unfair execution. Because we are left only with fair executions, we call refinement in FOS as *fair refinement*, as it only maps fair target behaviors to fair source behaviors. Then FOS develops a simulation that ensures fair refinement; given that traditional simulations prove refinement with stepwise rules, FOS also aims for a simulation providing such stepwise rules, without having to reason about the entire execution trace. The main obstacle in developing such a simulation is the very presence of FAIR, which decides the fairness based on the entire trace. We overcome this by developing stepwise simulation rules that are capable of dealing with the FAIR constructs directly, by correctly reflecting their semantics (filtering out unfair traces) in the simulation.

Fairness Counter. To reason about FAIR in our simulation rules, we introduce the concept of a **fairness counter**, which intuitively counts the number of bad events that happen before a good event. Formally, a fairness counter is a per-program (source / target) map whose domain is identical to the fairness map of the program (that is, the entities for which fairness must be ensured) and whose range is a set equipped with a well-founded relation. In this example, we take our fairness counter $\mathbf{c}$ to be of type $\texttt{cmap} : \{\mathsf{lot}\} \to \mathbb{N}$; *i.e.,* a map that assigns a natural number to lot.

The key idea behind the fairness counter is that each entity is mapped to a value which *cannot decrease indefinitely.* As stated, this counter counts the maximum number of bad events that an entity may see *before* encountering a good event: a value cannot decrease indefinitely under a well-founded relation, thus *limiting* the number of bad events that may happen to a finite value. The simulation keeps track of the fairness counter along with the continuation of the program, and will update the fairness counter when dealing with FAIRs in the source or target program—a bad event decreases the counter, while a good event will *reset* this counter to some arbitrary value.

Formally, we capture this update mechanism as a relation $\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}'$, and say that $\mathbf{c}'$ updates $\mathbf{c}$ given the fairness map $\mathbf{f}$ (the fairness map is required to determine the fairness ids for which the fairness counter should decrease and be reset). The exact definition of $\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}'$ depends on the set of entities considered; in this example, $\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}'$ can be defined as:

$$\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}' \triangleq \texttt{match}\ \mathbf{f}(\mathsf{lot})\ \texttt{with}\ \mid \texttt{good} \Rightarrow \top \mid \texttt{bad} \Rightarrow \mathbf{c}'(\mathsf{lot}) < \mathbf{c}(\mathsf{lot})$$

This definition states that $\mathbf{c}'$ updates $\mathbf{c}$ if (i) lot triggers good, in which case *any* $\mathbf{c}'$ updates $\mathbf{c}$, or (ii) lot triggers `bad`, in which case the counter of lot must *decrease*. Because the counter of lot cannot decrease below 0, updating the fairness counter ensures that unfair traces in which lot triggers `bad` infinitely are not considered when constructing our simulation; this corresponds to the semantics of `FAIR` which filters out unfair traces, allowing the fairness counter to capture the semantics of `FAIR` within a simulation.

Exploiting Fairness Now, we wish to construct a simulation that relies on `lottery`() being fair to prove that LOT[SPEC] refines WIN. The following rule enables this by dealing with `FAIR` in the target program when constructing a simulation:

$$\text{SIMFT}\quad \frac{\forall \mathbf{c}'.\ (\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}') \rightarrow (\mathbf{c}', k) \lesssim \mathbb{S}}{(\mathbf{c},\ \mathtt{FAIR}(\mathbf{f});\ k) \lesssim \mathbb{S}}$$

where k denotes the continuation, $\mathbb{S}$ the source, and the relation $\lesssim$ denotes that the right-hand argument of $\lesssim$ simulates the left-hand argument. Note that SIMFT (and our simulation rules in general) relates a *pair* of a fairness counter and a continuation. The rule consumes a `FAIR` from the *target*, and intuitively *applies the effect* of the `FAIR` construct by updating the fairness counter from $\mathbf{c}$ to $\mathbf{c}'$. In particular, SIMFT states that we *only need to consider* cases where $\mathbf{c}'$ updates $\mathbf{c}$ under $\mathbf{f}$: as we will illustrate below, this has the effect of *discarding* unfair executions of the target in the simulation, just as `FAIR` filters out unfair executions.

As an example, consider an application of SIMFT in order to prove the following simulation:

$$([\mathsf{lot} \mapsto i], \mathtt{FAIR}([\mathsf{lot} \mapsto \mathtt{bad}]);\ k) \lesssim \mathbb{S}$$

SIMFT applied to this proof goal requires that we prove a simulation for any $\mathbf{c}'$ that updates $[\mathsf{lot} \mapsto i]$ when lot triggers `bad`: that is, any $[\mathsf{lot} \mapsto i']$ for $i' < i$. If lot keeps on losing due to an unfair execution, the simulation proceeds until the following proof goal is reached:

$$([\mathsf{lot} \mapsto 0], \texttt{FAIR}([\mathsf{lot} \mapsto \texttt{bad}]); k) \lesssim \mathbb{S}$$

At this point, observe that there does *not exist* any $\mathbf{c}'$ such that $\mathbf{c}'$ updates $[\mathsf{lot} \mapsto 0]$, as $\nexists i' \in \mathbb{N}.\ i' < 0$. Thus the premise of this proof goal becomes a vacuous truth, and it follows that the simulation trivially holds for such unfair executions: this is the effect of SIMFT discarding unfair target executions in the simulation, similar to how FAIR filters out unfair executions!

On the other hand, suppose that lot encounters a good event and wishes to prove the following:

$$([\mathsf{lot} \mapsto i], \texttt{FAIR}([\mathsf{lot} \mapsto \texttt{good}]); k) \lesssim \mathbb{S}$$

In this case, the updated counter $\mathbf{c}'$ may be an *arbitrary* natural number: this ensures that fair traces are not discarded, no matter how many bad events precede a good event. Since a good event returns *true*, the simulation will then be able to match a *print* from LOT[SPEC] with a *print* from WIN, and further apply the same reasoning inductively to prove that LOT[SPEC] indeed does refine WIN.

The power of SIMFT allowing us to discard unfair traces of the target is what makes constructing a simulation that considers only fair behavior possible: we call this property **fairness exploitation**.

2.2.3 Proving that lottery() is Fair: Validating Fairness

To wrap this section up, we consider constructing a simulation in the opposite direction to §2.2.2: how can we deal with FAIRs in the *source* program? Such scenarios emerge when one is trying to prove that a certain implementation is indeed fair: *e.g.,* when proving that a specific implementation of `lottery()` refines SPEC.

Consider the following two implementations of `lottery`():

$$\texttt{def lot}_{\texttt{fair}}() \;=\; \texttt{if } x \texttt{ then } x := \mathit{false};\; \texttt{ret } \mathit{true} \texttt{ else } x := \mathit{true};\; \texttt{ret } \mathit{false}$$
$$\texttt{def lot}_{\texttt{unfair}}() = \texttt{ret } \mathit{false} \tag{IMPL}$$

Before discussing the fairness of $\texttt{lot}_{\texttt{fair}}$ and $\texttt{lot}_{\texttt{unfair}}$ directly, observe that both $\texttt{lot}_{\texttt{fair}}$ and $\texttt{lot}_{\texttt{unfair}}$ are trivially 'fair' in themselves as their executions are finite. However, when used with LOT, clearly $\texttt{lot}_{\texttt{unfair}}$ becomes unfair while $\texttt{lot}_{\texttt{fair}}$ is still fair. As illustrated, the *context* in which an implementation is used has an effect on whether the implementation is fair or not; we will thus prove that LOT[$\texttt{lot}_{\texttt{fair}}$] refines LOT[SPEC] in order to prove that $\texttt{lot}_{\texttt{fair}}$ is a fair implementation.

SIMFS is the rule for dealing with `FAIR`s from the source in the simulation ($\mathbb{T}$ denotes the target):

$$\text{SIMFS}\;\frac{\exists \mathbf{c}'.\ (\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}') \wedge \mathbb{T} \lesssim (\mathbf{c}', k)}{\mathbb{T} \lesssim (\mathbf{c}, \texttt{FAIR}(\mathbf{f}); k)}$$

Like SIMFT for consuming `FAIR`s in the target, SIMFS consumes `FAIR`s from the *source* and models the effect of `FAIR` in the simulation. In contrast to SIMFT, SIMFS states that there *must exist* a fair update of the fairness counter **c**: this ensures that there must exist a fair behavior of the *target* corresponding to the fair source behavior in order for the simulation to hold.

To see this effect, let us attempt to prove that LOT[$\texttt{lot}_{\texttt{unfair}}$] fairly refines LOT[SPEC] (which is clearly untrue). Assume this time that we are starting with the following proof goal, where `FAIR` occurs in the source LOT[SPEC]:

$$\mathbb{T} \lesssim ([\mathsf{lot} \mapsto i], \texttt{FAIR}([\mathsf{lot} \mapsto \mathsf{bad}]); k)$$

Observe that because the *target* LOT[$\texttt{lot}_{\texttt{unfair}}$] has a trace where it cannot print any "win!"s due to $\texttt{lot}_{\texttt{unfair}}$ returning *false* indefinitely, the source must also be able to match this trace for a simulation to hold. Clearly, lot must trigger bad indefinitely within this source-trace as well: thus the fairness map is fixed to $[\mathsf{lot} \mapsto \texttt{bad}]$, and applying SIMFS multiple times in this fashion will eventually again yield a proof goal where the fairness counter of lot is zero:

$$\mathbb{T} \lesssim ([\mathsf{lot} \mapsto 0], \texttt{FAIR}([\mathsf{lot} \mapsto \texttt{bad}]);\ k)$$

Here, observe that we *cannot* apply SIMFS to consume the source-FAIR anymore as there no longer exists a fairness counter that can update $[\mathsf{lot} \mapsto 0]$. In essence, the fairness counter limits the behavior of the *source* to be fair, similar to how FAIR in the source filters out unfair traces of the source. The fact that the target LOT[$\texttt{lot}_{\texttt{unfair}}$] has unfair behavior, which cannot be simulated by the only-fair behavior of the source, is reflected by SIMFS becoming no longer applicable when constructing the simulation. Because we can no longer consume the FAIR of the source, the simulation can no longer proceed and thus we cannot prove that LOT[$\texttt{lot}_{\texttt{unfair}}$] refines LOT[SPEC]!

In contrast, suppose now that we are correctly trying to prove that LOT[$\texttt{lot}_{\texttt{fair}}$] refines LOT[SPEC]. Because LOT[$\texttt{lot}_{\texttt{fair}}$] is fair and emits ”win!” every other iteration, the source program can also let lot trigger $\texttt{good}$ when constructing the simulation:

$$\mathbb{T} \lesssim ([\mathsf{lot} \mapsto i], \texttt{FAIR}([\mathsf{lot} \mapsto \texttt{good}]);\ k)$$

At this point, applying SIMFS allows us to update $\mathbf{c}'$ to any value; the simulation can thus choose a value larger than the number of bad events it will see before seeing another good event (here, even 1 will suffice as the target LOT[$\texttt{lot}_{\texttt{fair}}$] alternates between printing ”win!” and not). One can successfully prove that LOT[$\texttt{lot}_{\texttt{fair}}$] refines LOT[SPEC] by applying this reasoning inductively.

The reasoning principle enabled by SIMFS stands in direct contrast with the reasoning principle enabled by SIMFT, which allowed us to discard unfair target traces by corresponding them to vacuous truths. In contrast, SIMFS requires that the target must *only exhibit* fair behavior for it to be able to refine a (fair) source: we call this principle **fairness validation**.

2.3 Main Ideas: Concurrency and Thread-Local Reasoning

Having established the basic reasoning principles behind FOS, we now illustrate how fairness under concurrency with schedulers and locks can be modeled in FOS (§2.3.1), and then how refinement for concurrent programs can be proved in FOS, particularly in a thread-local manner (§2.3.2).

The running example in this section is identical to the motivating example from §2.1 (memory locations and variables in the paper, stated otherwise, are initialized to 0):

$$\begin{array}{l||l}
 & \texttt{do}\ \{ \\
\curlyvee;\ \texttt{lock}_{\texttt{abs}}(); & \quad \curlyvee;\ \texttt{lock}_{\texttt{abs}}(); \\
\curlyvee;\ X := 42; & \quad \curlyvee;\ x := X; \\
\curlyvee;\ \texttt{unlock}_{\texttt{abs}}(); & \quad \curlyvee;\ \texttt{unlock}_{\texttt{abs}}();\ \curlyvee; \\
\curlyvee; & \}\ \texttt{while}\ (x = 0) \\
 & \curlyvee;\ \mathit{print}(x);\ \curlyvee;
\end{array} \qquad (\mathrm{CL_I})$$

We assume sequential consistency as the memory model for $\mathrm{CL_I}$; later in §2.6.1, we will encode memory fairness to allow the use of weak memory models as well. As discussed, $\mathrm{CL_I}$, is a worker (thread 1) and a waiter (thread 2) process, which relies on a lock to protect non-atomic accesses to the shared memory location X. In this paper, we define the semantics of yielding such that a thread will yield if and only if it meets an explicit *yield instruction* $\curlyvee$ in the program: this models threads to be altruistic, and also has the effect of treating program fragments between $\curlyvee$s as atomic.

As discussed in §2.1, we wish to prove that $\mathrm{CL_I}$ fairly refines the following specification $\mathrm{CL_S}$:

$$\curlyvee;\ \texttt{skip};\ \curlyvee;\ \Big\|\ \curlyvee;\ \mathit{print}(42);\ \curlyvee; \qquad (\mathrm{CL_S})$$

To prove this refinement, one must be capable of establishing that thread 1 eventually acquires the lock, which in turn depends on fairness of the scheduler and the lock. We thus formally express scheduler and lock fairness first in §2.3.1, then illustrate the key argument required for establishing that thread 1 eventually acquires the lock during a fair execution in §2.3.2.

2.3.1 Fair Specifications for Concurrent Components

To prove that $\mathrm{CL_I}$ fairly refines $\mathrm{CL_S}$, one first requires a specification that encodes the fairness of the scheduler and the lock. In a concurrent setting, this means that fairness should be ensured for all threads, which are vying to get scheduled or acquire the lock. We can express these two distinct kinds of fairness by setting the fairness ids $\mathtt{ID}$ as the disjoint union of $\mathtt{th}_i$, the ids for scheduler fairness, and $\mathtt{lk}_i$, the ids for lock fairness, where i is a thread id. Thus $\mathtt{ID}$ becomes the domain of the fairness map $\mathtt{fmap}$ and the fairness counter $\mathtt{cmap}$:

$$\mathbf{f} \in \mathtt{fmap}_{\mathtt{ID}} \triangleq \mathtt{ID} \to \{\mathtt{good}, \mathtt{bad}\} \qquad \mathbf{c} \in \mathtt{cmap}_{\mathtt{ID}} \triangleq \mathtt{ID} \to \mathbb{N}$$

The events $\mathtt{good}$ and $\mathtt{bad}$ again model good and bad events: in this case, a $\mathtt{good}$ event is when a thread is scheduled or successfully acquires a lock.

Scheduler Fairness. In this paper, concurrency is modeled via thread interleaving, where a thread pool is a finite map from a thread id (in $\mathbb{N}$) to a code. Then the scheduler manages a thread pool by kicking out threads when they terminate; we call the remaining threads in the pool valid threads, and denote their ids as $\mathbb{N}_v$. Under this thread interleaving model, a scheduler may be viewed as a program that picks a valid thread id and executes the code of that thread; afterward, a yield $\curlyvee$ will return execution to the scheduler which will proceed to schedule the next thread.

A fair spec of a scheduler may be written as follows, where P represents the thread pool:

$$\lambda P.\ \texttt{while}(\mathbb{N}_v \neq \varnothing)$$
$$\{\ n \leftarrow \texttt{PICK}(\mathbb{N}_v);\ \texttt{FAIR}([\texttt{th}_n \mapsto \texttt{good},\ \{\texttt{th}_m \mid m \in \mathbb{N}_v \wedge m \neq n\} \mapsto \texttt{bad}]);$$
$$\text{exec}(P(n))\ \} \tag{SCH}$$

In SCH, `PICK` nondeterministically picks a valid thread id n to execute. The fairness of SCH is guaranteed by the following `FAIR` constructor, whose fairness map assigns `good` to the scheduled thread and `bad` to all other threads: following the semantics of the `FAIR` constructor, it is clear that any fair execution of SCH will result in valid threads becoming scheduled infinitely often. This captures exactly the concept of 'scheduler fairness' that programmers rely on when writing programs: that running threads will not be starved indefinitely by the scheduler.

Throughout the paper, we assume that schedulers satisfy SCH if not stated otherwise; this allows us to exploit scheduler fairness. We emphasize that SCH represents only the *fair* schedulers. Therefore, if a system's correctness relies on SCH, one should also validate the fairness of the scheduler implementation employed by the system to guarantee the full correctness of the system.

Lock Fairness. In addition to scheduler fairness, proving that CL_I refines CL_S requires fairness of the *lock*: clearly CL_I will fail to terminate if, e.g., the lock implementation is unfair and never grants the lock to thread 1. Encoding this fairness requirement once again requires a specification for the lock, which in addition to guaranteeing fairness as desired, should ideally abstract away implementation details such as data structures or memory accesses. The second desideratum allows different *concrete implementations* to refine the specification, as done in §2.2.3.

Fig. 2.2 presents such a specification for fair locks. Consider the lock function $\texttt{lock}_\texttt{abs}()$: here, each thread with thread id i is represented using a new entity

```
$W \subseteq \{\mathtt{lk}_i \mid i \in \mathbb{N}\}$    $own \in \mathbb{B}$
def lock$_{\mathtt{abs}}$() =
    $n$ = GetTid;  $W = W \cup \{\mathtt{lk}_n\}$;                                  //L1
    loop { if $own$ then $\curlyvee$;  else break }                          //L2
    $own = true$;  $W = W \setminus \{\mathtt{lk}_n\}$;  FAIR([$\mathtt{lk}_n \mapsto$ good, $W \mapsto$ bad])  //L3
    ret                                                                       //L4
def unlock$_{\mathtt{abs}}$() =
    assume($own = true$);  //L1
    $own = false$;         //L2
    ret                    //L3
```

Figure 2.2 A fair spec of fair locks, ABSLock.

$\mathtt{lk}_i$, which is a new fairness id for each thread required to ensure that a good obtained via a scheduling event does not also result in a good of the lock, and vice versa. The specification is straightforward: a thread will add itself to the waiting set W and enter the loop to wait for a lock, then trigger a good for itself and bads for all other threads in W upon acquiring a lock. This implies that a thread *waiting* for the lock must eventually acquire the lock if the lock is available, as it will accumulate an infinite number of bads otherwise (assuming scheduler fairness). On the other hand, a thread outside of the waiting set W is unaffected by the fairness events triggered by the lock: this captures the common concept of 'lock fairness', in that a thread attempting to acquire an available lock will eventually succeed.

```
($T$, $S$: globally fixed memory locations)
def lock$_{\mathtt{tk}}$()     $= t \coloneqq$ FAI($T$);  do { $s \coloneqq S$; } while($t \neq s$)     (TicketLock)
def unlock$_{\mathtt{tk}}$() $= s \coloneqq S$; $s = s + 1$; $S \coloneqq s$;
```

To conclude the discussion on fair concurrent specifications, let us briefly consider how the ticket lock implementation in TicketLock can validate the fair specification given in Fig. 2.2. TicketLock issues a ticket counter t to every thread that attempts to acquire the lock; this counter is guaranteed to strictly increase due to the *fetch-and-increment* instruction FAI(T), which increments the value stored at the memory location T by 1 and returns the old value. When a thread calls `unlock`$_{\texttt{tk}}$(), the service counter stored in the memory location S is increased: this allows the ticket lock to service the next thread that holds the ticket corresponding to the current service counter.

TicketLock is fair under sequential consistency and scheduler fairness, refining the fair specification from Fig. 2.2. This is because the ticketing scheme ensures that only a finite number of threads may be queued before a thread to acquire a lock; this means that a thread can only trigger a finite number of `bad`s before successfully acquiring the lock, ensuring that the trace is fair.

2.3.2 Exploiting Fairness Under Concurrency via Thread-Local Reasoning

Given the fairness properties we have encoded in §2.3.1, we now illustrate how one may actually prove that CL_{I} fairly refines CL_{S}. The key intuition in this section is that one may perform induction on the fairness counters when constructing a simulation to discard unfair behavior. Furthermore, these fairness counters may be treated as *shared state* between threads, which enables *thread-local reasoning* when constructing a simulation.

Global Proof by Induction. Before illustrating how thread-local reasoning can be performed in FOS, we first establish a global argument to prove that CL_{I} fairly refines CL_{S}; we will later show that this argument extends naturally to thread-local reasoning.

One of the main reasoning patterns enabled by fairness is the ability to argue that something `good` must happen eventually in a fair execution—*e.g.,* , thread 1 eventually acquires the lock in CL_{I}. In CL_{I}, this is required to ensure that the loop in thread 2 of CL_{I} eventually exits in a fair execution, which allows one to match the final *print* with the *print* in CL_{S} when constructing a simulation.

The key idea in proving that a `good` event happens is to construct a value that *decreases* but cannot decrease infinitely, until the `good` event is triggered.

The fairness counter **c** (from the global simulation relation in §2.2) captures perfectly this required decreasing value. Recall that the fairness counter also allows one to discard unfair executions of the target: thus by performing induction on the fairness counter, one can focus only on fair executions when constructing the simulation.

To see this strategy in detail, reconsider the fact that we must show thread 1 of CL_I eventually acquires the lock (a `good` event) when constructing a simulation. Intuitively, $\texttt{lk}_1$, the fairness counter for thread 1 acquiring the lock, can serve as the decreasing value for this `good` event, as thread 2 acquiring the lock instead will trigger a `bad` for $\texttt{lk}_1$. Then by performing induction on $\texttt{lk}_1$, one can establish that either (i) thread 1 acquires the lock and writes 42 to X, or (ii) the execution being considered is unfair, as $\texttt{lk}_1$ cannot decrease indefinitely.

To be more formal about the argument, one must also consider the fact that thread 2 progresses and eventually releases the lock (otherwise thread 1 will not accumulate a `bad`). This can be done by introducing an additional counter n, which keeps track of the remaining lines of code (*i.e.,* the number of remaining yields) until the lock is released. Of course, progress of thread 2 relies on thread 2 being scheduled—which can in turn be captured by the fairness counter $\texttt{th}_2$. Thus the *actual* decreasing value for thread 1 to acquire the lock is given as a tuple $(\texttt{lk}_1, n, \texttt{th}_2)$, where the order of the tuples is given by lexicographic order: $\texttt{lk}_1$ is the most significant as the subsequent two counters serve to ensure that $\texttt{lk}_1$ decreases. A formal proof would perform induction on this tuple to ensure progress as opposed to merely $\texttt{lk}_1$.

Thread-Local Reasoning for Fair Refinement. Given the aforementioned proof strategy based on induction on a decreasing value, we now show how this reasoning may be performed in a *thread-local* manner to prove that CL_I fairly refines CL_S without considering all thread interleavings.

As mentioned, the key idea that enables local reasoning for FOS is that the fairness counter—which captures the concept of fair behavior in the simulation—can naturally be treated as *shared state* between threads. This allows each thread to perform local reasoning via a **shared protocol**, in which threads *rely* on other threads upholding the protocol when resuming execution from a yield $\curlyvee$, and conversely *guaranteeing* that the protocol is satisfied before yielding. Through this rely-guarantee reasoning, one can perform induction thread-locally to prove that a refinement between threads holds using almost the same argument that was applied globally, *without* having to consider all thread interleavings in a global simulation.

In our example, consider the following protocol which states that threads satisfy one of the given three states on a yield. This protocol is used to ensure that thread 1 eventually acquires the lock:

- Either the lock is unowned $(\mathit{own} = \mathit{false})$,
- Thread 1 holds the lock, or
- Thread 2 holds the lock and the tuple $(\texttt{lk}_1, n, \texttt{th}_2)$ decreases.

Here, it is true that n is local to thread 2 and inaccessible to thread 1; we will temporarily assume that n is a 'ghost' variable that is managed by thread 2 on every yield for the sake of presentation.

Consider this protocol in the context of thread 2: if thread 1 does not hold the lock, then (i) thread 2 yields in a state where $\mathit{own} = \mathit{false}$ (directly after releasing the lock), or (ii) it will have decreased the tuple $(\texttt{lk}_1, n, \texttt{th}_2)$. This is because thread 2 acquiring the lock will either trigger a `bad` event for thread 1, or decrease n by progressing within the loop and reducing the number of remaining yields. Thus thread 2 *guarantees* that it upholds the protocol at each point it encounters a yield, *without* considering the behavior of thread 1 at all.

On the other hand, thread 1 may *rely* on the protocol being upheld when resuming execution at a yield point. If $\mathit{own} = \mathit{false}$, then thread 1 acquires the lock and the simulation can make progress. If *own* $=$ *true* but thread 1 does not have the lock, then thread 1 decreases the given tuple as $\mathtt{th}_2$ decreases due to thread 1 winning in the scheduler. Thus thread 1 can also *guarantee* that the protocol is upheld, without considering the behavior of thread 2.

As the shared protocol is guaranteed by both threads, one may perform induction locally in a thread to ensure progress. For example, in thread 1, applying induction using the protocol results in that either (*i*) thread 1 acquires the lock, or (*ii*) the execution is *unfair*, as the tuple failing to decrease indicates that the fairness counters $\mathtt{lk}_1$ or $\mathtt{th}_2$ cannot decrease, implying unfairness. Thus thread-local reasoning based on this protocol shows that executions where thread 1 cannot acquire the lock and does not terminate are unfair—and are thus discarded, allowing one to construct a thread-local simulation showing that thread 1 of CL_I fairly refines thread 1 of CL_S!

As shown above, protocols for thread-local reasoning about fairness often require the use of ghost values such as n, which may not be easy to expose to other threads. Fortunately, existing work on modern concurrent separation logic [12] allows us to express such protocols as invariants. FOS blends naturally with this idea, resulting in a simulation technique where true thread-local reasoning is enabled via separation logic; this technique is formalized in §2.7.

2.4 Core Definitions of FOS

We now formalize the ideas presented so far in Sections 2.4 to 2.6. In this section, we present (whole-program) fair semantics, refinement, and a simulation relation, which are straightforward formalizations of the ideas in §2.2; concurrency and the module system are formalized in §2.5 and §2.6.

2.4.1 Definitions of Fair Operational Semantics

To write fair programs (*e.g.,* those in §2.2), we first define a language `FL` (fair language) in Fig. 2.3. `FL` is coinductively defined with three constructors `FAIR`,

$$\texttt{ID} : \mathit{Type} \qquad \texttt{flag} \triangleq \{\texttt{good}, \texttt{bad}, \epsilon\} \qquad \texttt{fmap}_{\texttt{ID}} \in \texttt{ID} \to \texttt{flag} \qquad R : \mathit{Type}$$

$$\begin{aligned}
\texttt{FL}_{\texttt{ID},R} \overset{\text{coind}}{=} \;& \mid \texttt{FAIR}(\mathbf{f} \in \texttt{fmap}_{\texttt{ID}}) \gg\!= (k \in () \to \texttt{FL}) \\
& \mid \texttt{PICK}(X : \mathit{Type}) \gg\!= (k \in X \to \texttt{FL}) \\
& \mid \texttt{Obs}(\mathit{fn} \in \texttt{string}, \mathit{args} \in \texttt{list Val}) \gg\!= (k \in \texttt{Val} \to \texttt{FL}) \\
& \mid \texttt{ret}\ (r \in R) \mid \texttt{stuck}
\end{aligned}$$

$$\begin{aligned}
\text{SilEv} &\triangleq \mid \delta(\mathbf{f} \in \texttt{fmap}) \\
\text{ObsEv} &\triangleq \mid \text{obs}(\mathit{fn} \in \texttt{string}, \mathit{args} \in \texttt{list Val}, v \in \texttt{Val}) \\
\text{Trace} &\overset{\text{coind}}{=} \mid (e \in \text{SilEv} \uplus \text{ObsEv}) :: (\mathit{tr} \in \text{Trace}) \mid \text{Term}\ (r \in R) \mid \text{Error} \\
\text{Behavior} &\overset{\text{coind}}{=} \mid (o \in \text{ObsEv}) :: (\mathit{tr} \in \text{Behavior}) \\
& \qquad \mid \text{Diverge} \mid \text{Term}\ (r \in R) \mid \text{Error}
\end{aligned}$$

$$\begin{aligned}
&\text{FairTr} \in \mathbb{P}(\text{Trace}) \triangleq \{\, \mathit{tr} \mid \forall i \in \texttt{ID}.\, \text{FairTr}_i(\mathit{tr}) \,\} \\
&\text{FairTr}_{i \in \texttt{ID}} \in \mathbb{P}(\text{Trace}) \triangleq \nu X.\, \mu Y.\, \nu Z.\, \lambda \mathit{tr}. \qquad \textit{//}X, Y, Z \in \mathbb{P}(\text{Trace}) \\
&\texttt{match}\ \mathit{tr}\ \texttt{with} \mid \text{Term}\ r \Rightarrow \top \mid \text{Error} \Rightarrow \top \\
&\qquad \mid \text{obs}(\mathit{fn}, \mathit{args}, v) :: \mathit{tl} \Rightarrow Z(\mathit{tl}) \\
&\qquad \mid \delta(\mathbf{f}) :: \mathit{tl} \Rightarrow \\
&\qquad\quad \texttt{match}\ \mathbf{f}(i)\ \texttt{with} \mid \epsilon \Rightarrow Z(\mathit{tl}) \qquad \textit{//}Z\text{: inner coinductive} \\
&\qquad\qquad \mid \texttt{bad} \Rightarrow Y(\mathit{tl}) \qquad \textit{//}Y\text{: middle inductive} \\
&\qquad\qquad \mid \texttt{good} \Rightarrow X(\mathit{tl}) \qquad \textit{//}X\text{: outer coinductive}
\end{aligned}$$

Figure 2.3 Core definitions of the language `FL`, behavior, and fair trace.

`PICK`, and `Obs`, in addition to `ret` for termination and an explicit `stuck` to indicate an error, *i.e., undefined behavior*. The fairness constructor `FAIR` invokes fairness events via the fairness map `fmap`. `fmap`'s range is `flag`, which now includes ϵ to express that the fairness event is undefined for that index (usually omitted for brevity). `PICK` non-deterministically picks a value from any given set X, passing it to the continuation. Finally, `Obs` invokes an observable effect, *e.g.,* a system call, represented by a function name *fn* and arguments *args*, and passes the return value to the continuation. Terms in `FL` within the Coq development are defined using *interaction trees* [32], as opposed to directly embedded as Coq functions. This allows us to reuse programming constructs such as `match` … `with` for branches, $x \leftarrow p;\ k, \gg\!=$ for monadic bind, and `loop` for loops, thereby keeping our language minimal yet expressive.

The semantics of an `FL` program p is defined by the set of its possible *behaviors* Beh(p). For this, we first derive a set of possible *traces* from a program where a *trace* is a stream of silent (δ) or observable (obs) events and can possibly terminate with a return value or an error. The language construct $\texttt{FAIR}(\mathbf{f})$ emits $\delta(\mathbf{f})$, PICK emits $\delta([\_ \mapsto \epsilon])$, Obs emits corresponding obs, and $\texttt{ret}(r)/\texttt{stuck}$ terminates the trace correspondingly. Then, a behavior is defined as an observable summary of a trace in the sense that it drops all silent events and leaves only observable events. An infinite trace with only silent events results in a silent divergence [21].

Now, for a set of traces, we derive a set of behaviors by (i) filtering out *unfair* traces with help from δs, and (ii) erasing the now meaningless δs. A *fair* trace, defined by FairTr, is a trace that satisfies FairTr_i for every index i in ID. FairTr_i allows only finite bads until the next good for i, captured by the mixed coinductive-inductive-coinductive definition: the inner coinductive (Z) captures the possibly infinite fairness-irrelevant trace, the middle inductive (Y) ensures only finite bads are encountered until a good, which can appear infinitely as expressed by the outer coinductive (X).

Then whole-program refinement between two FL programs p_t and p_s is defined as follows:

$$p_t \sqsubseteq p_s \triangleq \text{Beh}(p_t) \subseteq \text{Beh}(p_s)$$

$\sqsubseteq$ is *transitive*, *reflexive*, and preserves termination (*i.e.,* if the target has (fair) divergence, so does the source). Note that the source and the target may have different IDs.

2.4.2 Simulation Relation

The simulation ($\lesssim$) presented in §2.2.2 is formalized in Fig. 2.4, which relates two FL programs. The rules are identical except that cmap is now parameterized over ID, the set of fairness indices, and $\texttt{C}_<$, a well-founded order.[2] Expectedly, the simulation satisfies the following adequacy theorem:

[2]We omit ID and $\texttt{C}_<$ for brevity whenever they are clear from the context.

$$\mathtt{cmap}_{\mathtt{ID},\,\mathtt{C}_<} \in \mathtt{ID} \to \mathtt{C}_< \qquad < \;\in \mathbb{P}(\mathtt{C}_< \times \mathtt{C}_<) \text{ is well-founded}$$

$$\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}' \in \mathbb{P}(\mathtt{cmap} \times \mathtt{fmap} \times \mathtt{cmap}) \triangleq \forall i \in \mathtt{ID}.$$

$$\mathtt{match}\; \mathbf{f}(i)\; \mathtt{with} \;\mid\; \mathtt{good} \Rightarrow \top \;\mid\; \mathtt{bad} \Rightarrow \mathbf{c}'(i) < \mathbf{c}(i) \;\mid\; \epsilon \Rightarrow \mathbf{c}'(i) = \mathbf{c}(i)$$

$$\lesssim\; \in \mathbb{P}((\mathtt{cmap} \times \mathtt{FL}) \times (\mathtt{cmap} \times \mathtt{FL}))$$

$$\text{SIMRET}\ \frac{r_t = r_s}{(\mathbf{c}_t,\, \mathtt{ret}\; r_t) \lesssim (\mathbf{c}_s,\, \mathtt{ret}\; r_s)} \qquad \text{SIMSTUCK}\ \frac{}{\mathbb{T} \lesssim (\mathbf{c},\, \mathtt{stuck})}$$

$$\text{SIMPT}\ \frac{\forall x \in X.\ (\mathbf{c},\, k(x)) \lesssim \mathbb{S}}{(\mathbf{c},\, \mathtt{PICK}(X) \gg\!= k) \lesssim \mathbb{S}} \qquad \text{SIMPS}\ \frac{\exists x \in X.\ \mathbb{T} \lesssim (\mathbf{c},\, k(x))}{\mathbb{T} \lesssim (\mathbf{c},\, \mathtt{PICK}(X) \gg\!= k)}$$

$$\text{SIMFT}\ \frac{\forall \mathbf{c}' \in \mathtt{cmap}.\ (\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}') \to (\mathbf{c}',\, k()) \lesssim \mathbb{S}}{(\mathbf{c},\, \mathtt{FAIR}(\mathbf{f}) \gg\!= k) \lesssim \mathbb{S}} \qquad \text{SIMFS}\ \frac{\exists \mathbf{c}' \in \mathtt{cmap}.\ (\mathbf{c} \hookrightarrow_{\mathbf{f}} \mathbf{c}') \land \mathbb{T} \lesssim (\mathbf{c}',\, k())}{\mathbb{T} \lesssim (\mathbf{c},\, \mathtt{FAIR}(\mathbf{f}) \gg\!= k)}$$

Figure 2.4 Definitions of `cmap` and selected rules of our simulation relation $\lesssim$ (slightly simplified).

Theorem 2.4.1 (Adequacy) *For a pair of programs p_t, p_s and a well-founded set $C_<$, we have:*

$$(\forall \mathbf{c}_t \in \mathit{cmap}_{(ID_t,\mathbb{N})}.\ \exists \mathbf{c}_s \in \mathit{cmap}_{(ID_s,\,S_<)}.\ (\mathbf{c}_t,\, p_t) \lesssim (\mathbf{c}_s,\, p_s)) \implies p_t \sqsubseteq p_s$$

2.5 Formalizing Fair Concurrency

In this section, we present the formal definitions of our model of concurrency and scheduler fairness following the ideas in §2.3.1, and show that a simple round-robin scheduler satisfies scheduler fairness.

2.5.1 Semantics of Concurrency

In this paper, concurrency is modeled by *interleaving* semantics with *thread-yields*, meaning that the scheduler interleaves the execution of the threads and each thread yields to the scheduler. To write concurrent programs, we define the thread language `TFL` and the scheduler language `SFL`. Then threads and a scheduler are together interpreted as a `FL` program, so the semantics (behavior) of a concurrent system (threads and a scheduler) is naturally defined by the semantics of `FL` (Fig. 2.5).

$$\begin{array}{l}
\mathtt{th} \triangleq \mathbb{N} \qquad R,\, \mathtt{ST} : \mathit{Type} \qquad \mathrm{TPool}_{\mathtt{ID},\, R,\, \mathtt{ST}} \triangleq \mathtt{th} \xrightharpoonup{\text{fin}} \mathtt{TFL}_{\mathtt{ID},\, R,\, \mathtt{ST}} \\
\mathit{Sch} \in \mathrm{scheduler}_R \;\triangleq (\mathtt{th} \times \mathrm{Fin}(\mathtt{th})) \to \mathtt{SFL}_R \\
\mathtt{TFL}_{\mathtt{ID},\, R,\, \mathtt{ST}} \overset{\text{coind}}{=} \mid \curlyvee \gg= (k \in () \to \mathtt{TFL}) \mid \mathtt{GetTid} \gg= (k \in \mathtt{th} \to \mathtt{TFL}) \\
\qquad\qquad\qquad \mid \mathtt{Put}(st \in \mathtt{ST}) \gg= (k \in () \to \mathtt{TFL}) \mid \mathtt{Get} \gg= (k \in \mathtt{ST} \to \mathtt{TFL}) \\
\qquad\qquad\qquad \mid \mathtt{FAIR}(\mathbf{f}) \gg= k \mid \mathtt{PICK}(X) \gg= k \mid \mathtt{Obs}(\ldots) \gg= k \\
\qquad\qquad\qquad \mid \mathtt{ret}\ r \mid \mathtt{stuck} \\
\mathtt{SFL}_{\mathtt{ID}=\mathtt{th},\, R} \overset{\text{coind}}{=} \mid \mathtt{Exec}(n \in \mathtt{th}) \gg= (k \in R? \to \mathtt{SFL}) \mid \mathtt{FAIR}(\mathbf{f}) \gg= k \mid \ldots
\end{array}$$

$$\begin{array}{l}
R^{\veebar} \triangleq \mid \blacktriangle(r \in R) \mid \veebar(t \in \mathtt{TFL}) \\
\mathrm{TI}(n \in \mathtt{th},\, t \in \mathtt{TFL},\, st \in \mathtt{ST}) \in \mathtt{FL}_{\mathtt{ID},\, R^{\veebar} \times \mathtt{ST}} \triangleq \\
\;\mathtt{match}\ t\ \mathtt{with} \mid \ldots \mid \mathtt{ret}\ r \Rightarrow \mathtt{ret}\ (\blacktriangle r,\, st) \\
\mid \curlyvee \gg= k \qquad\quad\;\; \Rightarrow \mathtt{ret}\ (\veebar k(),\, st) \\
\mid \mathtt{GetTid} \gg= k \;\; \Rightarrow \mathrm{TI}(n,\, k(n),\, st) \\
\mid \mathtt{Put}(st') \gg= k \;\; \Rightarrow \mathrm{TI}(n,\, k(),\, st') \\
\mid \mathtt{Get} \gg= k \qquad\;\; \Rightarrow \mathrm{TI}(n,\, k(st),\, st)
\end{array}$$

$$\begin{array}{l}
\mathrm{CI}(P,\, \mathit{Sch},\, st) \triangleq \mathrm{SI}(P,\, \mathit{Sch}(0,\, \mathtt{dom}(P) \setminus \{0\}),\, st) \\
\mathrm{SI}(P \in \mathrm{TPool},\, S \in \mathtt{SFL},\, st \in \mathtt{ST}) \in \mathtt{FL} \triangleq \\
\;\mathtt{match}\ S\ \mathtt{with} \mid \ldots \mid \mathtt{ret}\ r \Rightarrow \mathtt{ret}\ r \\
\mid \mathtt{Exec}(n) \gg= k \;\Rightarrow \mathtt{match}\ P(n)\ \mathtt{with} \mid \mathtt{None} \Rightarrow \mathtt{stuck} \\
\qquad \mid \mathtt{Some}\ t \Rightarrow x \leftarrow \mathrm{TI}(n,\, t,\, st);\ \mathtt{match}\ x\ \mathtt{with} \\
\qquad\quad \mid \blacktriangle r,\, st' \;\;\Rightarrow \mathrm{SI}(P[n \not\mapsto],\, k(\mathtt{Some}\ r),\, st') \\
\qquad\quad \mid \veebar t_k,\, st' \;\Rightarrow \mathrm{SI}(P[n \mapsto t_k],\, k(\mathtt{None}),\, st')
\end{array}$$

Figure 2.5 Definitions of the thread/scheduler language TFL/SFL and the interpreters CI, SI, TI.

Threads. We define a thread as a procedure in `TFL`, and a thread pool as a finite map from thread ids to `TFL`. `TFL` extends `FL` with *shared states* among threads (*e.g.,* memory) and concurrency features. The state `ST` is parameterized for flexibility, and `TFL` defines the `Get`/`Put` constructors to handle shared state. `TFL` also has constructors for concurrency, $\curlyvee$ (Yield) and `GetTid`: $\curlyvee$ enables a thread to yield to the scheduler, and `GetTid` returns the thread id of the current thread.

Scheduler. We define a scheduler as a *program* in `SFL`, parameterized by the initial thread id and a finite set of thread ids, Fin(`th`). `SFL` extends `FL` with the `Exec`(n) constructor, which *executes* the thread with id n. When a thread executes, it proceeds until it yields with $\curlyvee$ or terminates with `ret`, upon which control is returned to the scheduler; this process repeats until the thread pool is empty (or a thread gets `stuck`). One advantage of modeling the scheduler as a program is that we can implement various *scheduling policies* in `SFL`, and we can fix an *ideal* fair scheduler for an operational spec of fair schedulers.

Interpreting Concurrency. A concurrent system is *interpreted* into a `FL` program by the scheduler interpreter SI and the thread interpreter TI. The interpreters leave the common constructors (`FAIR`, `PICK`, `Obs`) as they are and only interpret the added constructors, such as `Get`, `Put`, and $\curlyvee$. We focus on the most important detail: how the thread interleavings are realized by the interpreters.

The thread interpreter TI enables thread interleaving using the *decorated return type* $R^{\veebar}$. It is a disjoint union of two cases, (i) termination with a value r ($\blacktriangle r$) and (ii) yield with a continuation t ($\veebar t$). By distinguishing the two cases, the scheduler can correctly update the thread pool, as described in the interpreting rule for `Exec`(n) in SI: First, the scheduler executes thread n and waits for it to yield, which returns a decorated value in $R^{\veebar}$ together with an

```
$\texttt{FAIRSch}(n \in \mathrm{th}, \mathit{ths} \in \mathrm{Fin}(\mathrm{th})) \in \mathsf{SFL} \triangleq$
  $\texttt{loop}\ \{\ x \leftarrow \texttt{Exec}(n);\ \texttt{match}\ x\ \texttt{with}$                        //L1
    | $\texttt{None}\ \Rightarrow n' \leftarrow \texttt{PICK}(\{m \mid m \in \mathit{ths} \cup \{n\}\});$     //L2
                 $\mathit{ths}' := \mathit{ths} \cup \{n\} \setminus \{n'\};$                                  //L3
                 $\texttt{FAIR}([n' \mapsto \texttt{good}, \mathit{ths}' \mapsto \texttt{bad}]);$              //L4
                 $n := n';\ \mathit{ths} := \mathit{ths}';$                                                    //L5
    | $\texttt{Some}\ r \Rightarrow \texttt{if}\ \mathit{ths} = \varnothing\ \texttt{then ret}\ r$              //L6
                 $\texttt{else}\ n' \leftarrow \texttt{PICK}(\{m \mid m \in \mathit{ths}\});$                  //L7
                 $\mathit{ths}' := \mathit{ths} \setminus \{n'\};$                                             //L8
                 $\texttt{FAIR}([n' \mapsto \texttt{good}, \mathit{ths}' \mapsto \texttt{bad}]);$              //L9
                 $n := n';\ \mathit{ths} := \mathit{ths}'\ \}$                                                 //L10
```

```
$\texttt{FIFOSch}(n \in \mathrm{th}, \mathit{ths} \in \mathrm{Fin}(\mathrm{th})) \in \mathsf{SFL} \triangleq$
  $q := \texttt{set2queue}(\mathit{ths});$
  $\texttt{loop}\ \{\ x \leftarrow \texttt{Exec}(n);\ \texttt{match}\ x\ \texttt{with}$
    | $\texttt{None}\ \Rightarrow (n, q) := \texttt{pop}(\texttt{push}(n, q));$
    | $\texttt{Some}\ r \Rightarrow \texttt{if}\ q = []\ \texttt{then ret}\ r$
                 $\texttt{else}\ (n, q) := \texttt{pop}(q);\ \}$
```

$\texttt{set2queue} \in \mathrm{Fin}(\texttt{th}) \rightarrow \texttt{queue th}$
$\texttt{pop} \in \texttt{queue}\ A \rightarrow A \times \texttt{queue}\ A$
$\texttt{push} \in A \times \texttt{queue}\ A \rightarrow \texttt{queue}\ A$

Figure 2.6 Definition of a spec of fair schedulers `FAIRSch` and a simple round-robin scheduler `FIFOSch`.

updated shared state. Then the scheduler inspects the decorated value to (i) kick out a terminated thread ($\blacktriangle$) or (ii) update the thread pool with the continuation of the yielded thread ($\veebar$). Therefore, the scheduler tracks the continuation of each thread, being able to interleave the execution of threads in the pool. All in all, the concurrency interpreter CI interprets a concurrent system into a `FL` program.

2.5.2 Scheduler Fairness, Operationally

Scheduler fairness guarantees that every thread *eventually* gets scheduled infinitely often. We define scheduler fairness operationally by defining a fair specification for a scheduler `FAIRSch` (Fig. 2.6), and say that a scheduler guarantees scheduler fairness when it refines `FAIRSch`.

A Spec for Fair Schedulers. `FAIRSch` abstracts what one would expect from a scheduler through nondeterminism (`PICK`) and mathematical sets: it is a loop that terminates when all the threads have terminated (L6), executing a single thread each iteration. What makes it an *abstract and fair spec* is the fact that it schedules threads *non-deterministically*, hiding implementation details such as queues (L2, L7); and ensures fairness by invoking `bad` for the unscheduled threads (L4, L9).

We now present our formalization of *scheduler fairness*:

Definition 2.5.1 (Scheduler Fairness) *We say a scheduler Sch guarantees* scheduler fairness *when* ***IsFairSch*** *(Sch) holds.* ***IsFairSch*** *is defined as follows:*

$$\mathtt{IsFairSch}(Sch) \triangleq \forall\, P\, st.\ \mathrm{CI}(P,\, Sch,\, st) \sqsubseteq \mathrm{CI}(P,\, \mathtt{FAIRSch},\, st)$$

Example. To illustrate a concrete scheduler that guarantees scheduler fairness, we present a simple round-robin scheduler `FIFOSch` (Fig. 2.6). This scheduler uses a *queue* to schedule the threads in a first-in-first-out manner. Since every thread always gets scheduled after other threads in the thread pool get scheduled, which happens only finitely many times, it is clear that this scheduler guarantees fairness. We formally establish such a guarantee using the above definition:

Theorem 2.5.2 (`FIFOSch` is Fair) *IsFairSch (FIFOSch) holds.*

This theorem states that `FIFOSch` fairly refines `FAIRSch`, which corresponds to fairness validation: we wish to show that `FIFOSch` satisfies the fairness requirements set by `FAIRSch` using simulation. Thus the key rule application in the proof of Theorem 2.5.2 becomes SIMFS presented in §2.2.3, and the main challenge is to find a suitable value of $\mathbf{c}'$ to update the fairness counter with. In the case of `FIFOSch`, $\mathbf{c}'$ can be set to the maximum number of threads, denoted M: intuitively, the FIFO queue cannot contain more than M threads whenever a thread is enqueued. Thus a thread scheduled via `FIFOSch` can only trigger

$$\begin{array}{l}
\texttt{OTFL}_{\mathrm{ID},\,R,\,\texttt{ST}} \overset{\text{coind}}{=} \\
\;|\; \texttt{Call}(fn \in \texttt{string},\, args \in \texttt{list Val}) \gg\!= (k \in \texttt{Val} \to \texttt{OTFL}) \;|\; \ldots \\
M \in \mathrm{Mod}_{\mathrm{ID},\,\texttt{ST}} \triangleq \\
\;\{(\texttt{init},\, \texttt{funs}) \in \texttt{ST} \times (\texttt{string} \overset{\mathsf{fin}}{\rightharpoonup} (\texttt{list Val} \to \texttt{OTFL}_{\texttt{ID},\,\texttt{Val},\,\texttt{ST}}))\} \\
\mathrm{Config} \triangleq \texttt{th} \overset{\mathsf{fin}}{\rightharpoonup} (\texttt{string} \times \texttt{list Val}) \qquad \texttt{Load} \in \mathrm{Config} \to \mathrm{Mod} \to \mathrm{TPool} \\
\hline
\lesssim_m \in \mathbb{P}(\mathrm{Mod} \times \mathrm{Mod}) \\
M_1 \circ M_2 \in \mathrm{Mod}_{\texttt{ID}_1 + \texttt{ID}_2,\, \texttt{ST}_1 \times \texttt{ST}_2} \qquad M_1[M_2] \in \mathrm{Mod}_{\texttt{ID}_1 + \texttt{ID}_2,\, \texttt{ST}_1 \times \texttt{ST}_2} \\
M_1 \circ M_2 \lesssim_m M_2 \circ M_1 \qquad M_1 \lesssim_m M_2 \to M \circ M_1 \lesssim_m M \circ M_2 \\
M_t \lesssim_m M_s \to M_c[M_t] \lesssim_m M_c[M_s] \\
\hline
\end{array}$$

Figure 2.7 Definitions of our module system and properties of the module simulation.

less than M `bad` events within the proof before it gets scheduled and triggers a `good` event: using M as the value of $\mathbf{c}'$ when applying SIMFS captures this intuition.

2.6 Module System and Fair Specs

For reusability and modularity, FOS provides a module system (Fig. 2.7) inspired by [8] that comprises (*i*) module type Mod, consisting of module-local state and (possibly open) module functions, (*ii*) linking operation $\circ$ for modules, and (*iii*) close operation $M_1[M_2]$ that closes open functions in a module M_1 upon getting a module M_2. Open module functions are written in `OTFL`, which extends `TFL` with `Call`; the close operation $M_1[M_2]$ interprets $\texttt{Call}(fn, args)$s in M_1 by substituting them with the corresponding `TFL`s in M_2. Also, we define a configuration Config, which maps each thread id to a function name and arguments. `Load` takes a configuration and a module, initiates `TFL` for each thread by the function name and arguments, and outputs a thread pool.

What underlies the power of modularization provided by our module system is the module simulation $\lesssim_m$. Module simulation requires the user to prove *thread-local simulation* (see §2.7) for each pair of functions within the source-target modules, and establishes the refinement. Since the module linking and close operations respect $\lesssim_m$ (Fig. 2.7), this result implies *contextual refinement.*

Theorem 2.6.1 (Adequacy of Module Simulation) *For a pair of modules M_t and M_s, if $M_t \lesssim_m M_s$ holds, then for any configuration $p \in$ Config, refinement under scheduler fairness holds:*

$$\mathrm{CI}(\texttt{Load}\ p\, M_t,\ \textit{\textbf{FAIRSch}},\ M_t.\textit{\textbf{init}}) \sqsubseteq \mathrm{CI}(\texttt{Load}\ p\, M_s,\ \textit{\textbf{FAIRSch}},\ M_s.\textit{\textbf{init}})$$

2.6.1 Memory Modules and Memory Fairness

One key example of the module system is *memory modules* inspired by [28]. Defining memory models as modules grants us the flexibility required to instantiate concurrent programs with different kinds of memory models, *e.g.,* with a SC memory module *or* a weak memory module.

Then, as discussed in §2.3.1, one can prove that the ticket lock module under either SC / weak memory refines the spec. However, proving that the refinement holds under weak memory requires an extra fairness assumption: *memory fairness* [15]. This is because, under weak memory, a value written to memory must be propagated for other threads to read it (*e.g.,* by a cache coherence protocol)—which is not always guaranteed: a thread may never read the most recent update under a buggy protocol, which may render a thread unable to acquire a lock because it cannot read the most recent service value! Memory systems that rule out such cases are said to guarantee memory fairness, and threads eventually obtain newer values under memory fairness.

To model memory fairness in FOS, we develop a fair weak memory module `FWMM` under *view semantics*, a "promise-free" fragment of the promising semantics [13, 19] without fences, which can be also seen as a fragment of an operational version of RC11 [16][3]. In view semantics, each memory location has a *history* of written values with *timestamps.* Each thread has its own *view of memory*; intuitively, the view points to the most recently propagated value to that thread and increases monotonically following execution. A thread can only read the same or newer values than its current view, and can write to memory with newer timestamps than the current view. The following shows an example

[3]In the Coq development, we adopt the model developed in [3].

memory view for some thread k and location X:

$$\begin{array}{rccccccccc} X: & \boxed{v_0} & \boxed{v_1} & \boxed{v_2} & \boxed{v_3} & \boxed{v_4} & \boxed{v_5} & \boxed{v_6} & \dots & \boxed{v_n} \\ \text{timestamp}: & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & \dots & t_n \end{array}$$

In the figure, k's current view for X is t_2 and there are newer values that have not yet propagated to k (t_3 ˜ t_n). When thread k accesses X, it can update its timestamp to one of t_2 to t_n.

Based on the figure, we demonstrate how we encode memory fairness in our model. Suppose that thread k reads from X, which resulted in updating its timestamp at X from t_2 to t_4. At this moment, `bad` is invoked for every unpropagated timestamp, *i.e.*, $\texttt{FAIR}(\{t_i \mid 5 < i \leq n\} \mapsto \texttt{bad})$ is triggered. This guarantees that every timestamp is eventually propagated to the thread k upon continuous access to X since infinite continuous `bad` is unfair.

Returning to the ticket lock example from §2.3, we can prove that a ticket lock refines `ABSLock` thanks to the fairness guarantee of `FWMM`:

Theorem 2.6.2 (TicketLock is Fair) $\text{TicketLock}_{\texttt{FWMM}}$*, a ticket lock module under FWMM, is simulated by the spec ABSLock:* $\text{TicketLock}_{\texttt{FWMM}} \lesssim_m \texttt{ABSLock}$*. Thus it contextually refines ABSLock.*[4]

2.6.2 Expressing Various Concepts of Fairness

So far, we have demonstrated that FOS can model fair semantics of various systems, such as concurrency, library specifications, and memory models. However, the expressiveness of FOS reaches much further, being capable of describing various concepts of fairness proposed in the literature. We provide three examples, all assuming scheduler fairness: starvation/deadlock freedom, strong/weak fairness, and readers-writers problem.

Starvation/Deadlock Freedom. Starvation/Deadlock free concurrent objects guarantee certain progress of threads accessing the object: *starvation* free-

[4]This `ABSLock` needs slight modifications from the one under an SC memory, since we need to pass around the view. However, this detail is solely due to the view semantics, orthogonal to our discussion regarding fairness.

dom guarantees that *every* thread eventually makes progress, and *deadlock* freedom guarantees that *some* thread eventually makes progress [9]. We can express these properties in FOS, as illustrated by the following (X++ is executed atomically):

```
def incr_max() = ret X++

def incr_min() =
  while(PICK(𝔹)) { FAIR([$\alpha \mapsto$ bad]); $\curlyvee$; }
  FAIR([$\alpha \mapsto$ good]);  ret X++
```

(INCR)

One can easily see that $\texttt{incr}_{\texttt{max}}()$ guarantees *starvation freedom*, since any thread calling it immediately returns. More interesting is $\texttt{incr}_{\texttt{min}}()$, which guarantees *deadlock freedom* by imposing fairness to single index α: if every callee thread gets stuck in the loop, $[\alpha \mapsto \texttt{bad}]$ accumulates indefinitely. Hence by fair semantics, *some* thread will eventually get out of the loop, returning with the wanted value. However, that thread invokes a $[\alpha \mapsto \texttt{good}]$ before returning, which *resets* the accumulated up bads; so other threads stuck in the loop are *not* guaranteed to escape the loop.

Strong/Weak Fairness. Strong/Weak fairness is about progress of threads under constraints: *strong* fairness guarantees that "every thread that is *enabled infinitely often* gets its turn infinitely often", while *weak* fairness guarantees that "every thread that is *continuously enabled* gets its turn infinitely often" [1]. In general, this means that threads can make progress (*cf.* gets its turn) only when some condition is satisfied (*cf.* enabled). Then, each constraint can be expressed in FOS as the following (X-- is executed atomically):

```
def decr_st() =
  $n$ = GetTid;  $W = W \cup \{n\}$;
  while($X \le 0$) { $\curlyvee$; } $W = W \setminus \{n\}$;
  FAIR([$n \mapsto$ good, $W \mapsto$ bad]); ret X--

def decr_wk() =
  while($X \le 0$) { $\curlyvee$; } ret X--
```

(DECR)

These functions guarantee that they (*i*) decrement X and return if some increment function (*e.g.*, INCR) increments X to a nonnegative number, or (*ii*) loop infinitely—they are *enabled* only when $X > 0$. Then it is easy to see that `decr`$_{\texttt{wk}}$() guarantees *weak* fairness: if $X > 0$ remains true (*e.g.,* the increment function is always called in between) from some point, namely *continuously enabled*, every thread escapes the loop and returns. However, if X becomes 0 infinitely often, the thread may not be able to escape the loop. On the other hand, `decr`$_{\texttt{st}}$() guarantees *strong* fairness: if $X > 0$ is true *infinitely often* (*e.g.,* the increment function is called infinitely often), some thread escapes the loop, decreasing X. The escaping thread also invokes $[W \mapsto$ `bad`$]$ for the threads waiting to get enabled, thereby ensuring that any waiting thread eventually gets its turn. Note that `ABSLock` also guarantees strong fairness since the lock is eventually acquired if it is freed infinitely often; one can observe that the codes have a similar pattern.

The Readers-Writers Problem. The Readers-Writers problem is the problem of the mutual exclusion of several threads accessing a shared resource, where "readers" share the resource with other readers and "writers" require exclusive access [4]. We simplify the problem to one in which there is a single shared location X, where readers read from and writers increment. There are two kinds of problems: Problem 1 states that *readers should not be blocked unless a writer is writing.* Using FOS, we can specify an abstract spec that captures problem 1:

```
def read$_1$() =
    $n$ = GetTid;  $R = R \cup \{n\}$;
    while(PICK($\mathbb{B}$)) { FAIR([$n \mapsto$ bad]); $\curlyvee$; }
    $R = R \setminus \{n\}$;  FAIR([$n$, $W \mapsto$ good]);  ret $X$
def write$_1$() =                                                    (P1)
    $n$ = GetTid;  $W = W \cup \{n\}$;
    while(PICK($\mathbb{B}$)) { FAIR([$n \mapsto$ bad]); $\curlyvee$; }
    $W = W \setminus \{n\}$;  FAIR([$n$, $R \mapsto$ good]);  $X$++;  ret
```

The spec employs two sets to hold reader(R)/writer(W) threads, and those threads wait in a loop, invoking a `bad` for itself for each iteration. What makes this spec interesting is the `good` events, each invoked by the *opposite* class; *readers* invoke $[W \mapsto$ `good`$]$ and *writers* invoke $[R \mapsto$ `good`$]$. Intuitively, this means that each class *interrupts* the other class from making progress, since a `good` *resets* the `bad`s. Therefore, if readers are holding the resource, every reader will eventually make progress while any writers are blocked, and vice versa when a writer is holding the resource.

On the other hand, problem 2 states that *writers should write as soon as possible*. In other words, writers have priority over readers in accessing the resource. As with the previous case, we can specify an abstract spec that captures problem 2:

```
def read$_2$() =
  $n$ = GetTid;  $R = R \cup \{n\}$;
  while(PICK($\mathbb{B}$)) { FAIR([$n \mapsto$ bad]); $\curlyvee$; }
  $R = R \setminus \{n\}$;  FAIR([$n \mapsto$ good]);  ret $X$

def write$_2$() =
  FAIR([$R \mapsto$ good]);  $X$++;  ret
```

(P2)

This time, any writer should be able to write as soon as it wants, so it is not blocked by anyone. However, readers can still be blocked by the writers; thus all writers invoke $[R \mapsto$ `good`$]$, where R is the set of blocked readers, to *reset* the accumulated `bad`s of the readers. Note that [27] specifies these kinds of properties (some operations can prevent termination of others while other operations do not) under the name “impedance”.

2.7 A Program Logic for Fairness

In this section, we present a more *modular* and *abstract* interface to the simulation technique presented in §2.3, which we call **Fairness Logic**.

Fairness logic tackles two issues with the rudimentary simulation technique presented in §2.3.2. First, while the rudimentary technique achieves mostly thread-local reasoning by condensing all the needed “global” information in the form of `cmap`, the `cmap` itself remains a global object and reasoning around it remains global. Second, the proof of fair refinement often depends on conditions on ghost variables as shown in §2.3.2.

Fairness logic addresses these issues with the help of *separation logic*. Mod-

Rules for source FAIR

$$\trianglerighteq(i \in \mathtt{ID},\ o \in \mathrm{Ordinal})$$

⊵-SEP

$$\trianglerighteq(i, o_0 \oplus o_1) \dashv\vdash \trianglerighteq(i, o_0) * \trianglerighteq(i, o_1)$$

MONO

$$(o \geq o') * \trianglerighteq(i, o) \vdash \dot{\Rrightarrow} \trianglerighteq(i, o')$$

WIN-SRC

$$\frac{\trianglerighteq(i, o) \mathrel{-\!\!*} sim_I(Q,\ k_s,\ k_t)}{sim_I(Q,\ \mathtt{FAIR}([i \mapsto \mathtt{good}]);\ k_s,\ k_t)}$$

LOSE-SRC

$$\frac{\trianglerighteq(i, 1)\ *\ sim_I(Q,\ k_s,\ k_t)}{sim_I(Q,\ \mathtt{FAIR}([i \mapsto \mathtt{bad}]);\ k_s,\ k_t)}$$

Rules for target FAIR

$$\blacklozenge_{q \in (0,\,1]}(i \in \mathtt{ID},\ n \in \mathbb{N}) \quad \lozenge(i \in \mathtt{ID}) \quad \lozenge_{\mathtt{th}} \triangleq \forall i.\ \lozenge(\mathtt{th}_i) \quad \blacklozenge(i) \triangleq \exists n.\ \blacklozenge_1(i,\ n)$$

◆-SEP

$$\blacklozenge_{q_0+q_1}(i, min(n_0, n_1)) \dashv\vdash \blacklozenge_{q_0}(i, n_0) * \blacklozenge_{q_1}(i, n_1)$$

DEC

$$\frac{\blacklozenge_q(i, n)\ *\ \lozenge(i)}{\dot{\Rrightarrow} \exists n'.\ \blacklozenge_q(i, n')\ *\ (n' < n)}$$

LOSE-TGT

$$\frac{\lozenge(i) \mathrel{-\!\!*} sim_I(Q,\ k_s,\ k_t)}{sim_I(Q,\ k_s,\ \mathtt{FAIR}([i \mapsto \mathtt{bad}]);\ k_t)}$$

WIN-TGT

$$\frac{\blacklozenge(i)\ *\ (\blacklozenge(i) \mathrel{-\!\!*} sim_I(Q,\ k_s,\ k_t))}{sim_I(Q,\ k_s,\ \mathtt{FAIR}([i \mapsto \mathtt{good}]);\ k_t)}$$

Rules for $\curlyvee$

YIELD-SRC

$$\frac{sim_I(Q,\ k_s,\ \curlyvee;\ k_t)}{sim_I(Q,\ \curlyvee;\ k_s,\ \curlyvee;\ k_t)}$$

YIELD-TGT

$$\frac{b \in \mathcal{B} \qquad I * (I \mathrel{-\!\!*} \blacklozenge(\mathtt{th}_{tid}) * (\blacklozenge(\mathtt{th}_{tid}) \mathrel{-\!\!*} \lozenge_{\mathtt{th}} \mathrel{-\!\!*} sim_I(Q,\ (b\,?\ \curlyvee\ :\ skip);\ k_s,\ k_t)))}{sim_I(Q,\ \curlyvee;\ k_s,\ \curlyvee;\ k_t)}$$

Figure 2.8 Core rules of fairness logic.

ern separation logic[5] solves the first issue by allowing modular reasoning on global state via the notion of ownership and ownership transfer. Separation logic also allows one to introduce user-defined ghost variables conforming to certain protocols defined via the theory of Partial Commutative Monoids (PCM), solving the second issue: our examples indeed reuse a large body of theory previously developed around PCMs [11].

2.7.1 Core Rules of Fairness Logic

Core rules of fairness logic are presented in Fig. 2.8, comprising: rules for (i) executing `FAIR()` in the source, (ii) executing `FAIR()` in the target, and (iii) executing $\curlyvee$.

Simulation Weakest Precondition A central notion in our rules is "simulation weakest precondition" [6]: $sim_I(Q,\ k_s,\ k_t)$ denotes the weakest precondition to simulate k_t (target) against k_s (source) with postcondition Q, under a relational invariant I shared among threads. In the reasoning, one can *rely* on that I holds when it receives control (*i.e.,* at the beginning of the function and after $\curlyvee$) and should *guarantee* that I holds when it transfers control (*i.e.,* at the end of the function and before $\curlyvee$). Our simulation weakest precondition satisfies all the standard rules for simulation argument and weakest precondition, which we omit here.

Rules for executing source `FAIR` We start by presenting the fairness assertions used by the source rules. For the source fairness counters, we provide $\trianglerighteq(i,\ o)$ denoting the *right* (or, ownership) to decrement the counter of id i by an ordinal o: this assertion is *local* as it only concerns the id i instead of the global `cmap`. Also, the $\trianglerighteq$-SEP rule gives equivalence between $\trianglerighteq(i, o_0 \oplus o_1)$ and its decomposition $\trianglerighteq(i, o_0) * \trianglerighteq(i, o_1)$ where $\oplus$ is the natural sum [10] of ordinals. Such a split resource can then be distributed across threads, ensuring local

[5]We use a non-step-indexed variant of Iris [12] resource algebra, developed in CCR [28].

reasoning even when multiple threads are accessing the same i. $\trianglerighteq$ also enjoys monotonicity on its second argument (MONO).

Now, the WIN-SRC rule says that one gets $\trianglerighteq(i, o)$ for o of its choice when winning. Such a rule is designed with stress on usability: it does not require *any* $\trianglerighteq$ as a precondition. This in turn means $\trianglerighteq$ in your frame remains valid after the rule, and this is still sound because under the hood it *increments* the counter by o, instead of setting it exactly into o. The LOSE-SRC rule says that it consumes a right to decrement i by one, and actually decrements the counter by one under the hood. Note how these rules together imposes **fairness validation** as expected.

Rules for executing target `FAIR` Rules for executing target `FAIR` follow a similar spirit to those for the source. Nonetheless, the rules are not exactly symmetric because of a different nature between fairness validation and fairness exploitation.

Fairness assertions that the target rules will use are twofold: (i) $\blacklozenge_q(i, n)$ for an id i, a fraction q, and a natural number n denotes knowledge that the counter for i is at most n and a fractional ownership q to execute `good`, and (ii) $\lozenge(i)$ for an id i denotes a "receipt" for decreasing the counter for i by one. As before, we have a rule ($\blacklozenge$-SEP) decomposing $\blacklozenge$ but with a twist: since n does not refer to the value of the counter but instead means the upper bound for the counter, n does not get split into two when splitting and we take the minimum of the two ns when merging two $\blacklozenge$s. The most interesting rule in the target side is the DEC rule, concerning **fairness exploitation**: it consumes a $\blacklozenge$ and a $\lozenge$ to produce a $\blacklozenge$ with a decreased number. Such a rule coincides with intuitive interpretation of $\blacklozenge$ and $\lozenge$, and allows fairness exploitation since a number cannot decrease indefinitely.

Now, the LOSE-TGT rule says that one gets $\lozenge(i)$ when id i triggers a bad event. Again, such a rule is designed with usability in mind: a direct, lower-level reasoning would dictate the exact counter value for i before/after execution, but would not be ideal for local reasoning (*e.g.,* when multiple threads trigger bad events on the same id, some synchronization has to be made to track the latest value). Here the rule requires *nothing* in the precondition: this is because it does not need the exact value of i, but only the fact that a decrement has been made. The WIN-TGT rule consumes the full fraction of $\blacklozenge$ with any value n and returns the full fraction of $\blacklozenge$ with an unknown value n. Note the difference with WIN-SRC: the counter value for i is updated to an unknown value, and the rule contains the full fraction in the precondition, which is required for soundness.

Rules for executing $\curlyvee$ Finally, we give rules about executing $\curlyvee$ that are capable of reasoning about *scheduler fairness* in a thread-local fashion.

The YIELD-SRC rule allows executing $\curlyvee$ in the source as if it is "skip", in the presence of the $\curlyvee$ in the target. Now, consider scheduler fairness: the rule needs to somehow impose fairness validation regarding scheduler fairness events. For this, instead of baking in WIN-SRC and LOSE-SRC rules into this rule as-is, we give a much simpler yet expressive enough interface: following Simuliris [6], we use an inductive-coinductive definition that allows using the YIELD-SRC rule only finitely unless coinductive progress is made in the YIELD-TGT rule (below). This allows, in the majority of verification scenarios, the user to completely ignore proof obligations regarding (scheduler) fairness validation.

On the other hand, the YIELD-TGT rule executes $\curlyvee$ in the target. When b is false, it simply executes $\curlyvee$ on both sides in lock-step with the usual rely-guarantee principle on I (it demands I to hold before and gives it back after). When b is true, it just executes $\curlyvee$ in the target (keeping $\curlyvee$ in the source): this flexibility is useful as it allows one to execute multiple target $\curlyvee$s with a single source $\curlyvee$. Now, consider scheduler fairness. Recall that after the current thread, *tid*, takes the control back, a fairness event for winning *tid* and losing everyone else is invoked. The rule has applications of WIN-TGT and LOSE-TGT to this event baked-in, which appears as $\blacklozenge(\mathtt{th}_{tid})$ and $\lozenge_{\mathtt{th}}$. $\lozenge_{\mathtt{th}}$ is simply a conjunction of $\lozenge$ for *all* possible thread ids.

Adequacy Fairness logic satisfies the following two adequacy theorems.

Theorem 2.7.1 (Contextual Adequacy) *For a pair of module M_t, M_s, and a relational invariant I, if the initial states of the modules satisfy I, we have:*

$$(\forall\, tid\, f\, v.\ I * \blacklozenge(\mathit{th}_{tid}) \vdash \mathit{sim}_I(I * \blacklozenge(\mathit{th}_{tid}),\ M_s.\mathit{funs}\, f\, v, M_t.\mathit{funs}\, f\, v)) \implies M_t \lesssim_m M_s$$

Here, $I * \blacklozenge(\mathtt{th}_{tid})$ plays essentially the same rely-guarantee reasoning with the YIELD-TGT rule.

Theorem 2.7.2 (Whole Program Adequacy) *For a pair of modules M_t, M_s, a relational invariant I, a whole-program configuration $p \in$ Config, and a precondition P_{tid} for each thread id tid, if the initial states of the modules satisfy $\mathop{\Large *}_{tid \in \mathit{dom}(p)} \blacklozenge(\mathit{th}_{tid}) \twoheadrightarrow (I * \mathop{\Large *}_{tid \in \mathit{dom}(p)} P_{tid})$, we have:*

$$(\forall\, tid\, f\, v.\ (p\ tid = (f, v)) * I * P_{tid} \vdash \mathit{sim}_I(I * \blacklozenge(\mathit{th}_{tid}),\ M_s.\mathit{funs}\, f\, v, M_t.\mathit{funs}\, f\, v))$$
$$\implies$$
$$\mathrm{CI}(\mathtt{Load}\ p\, M_t,\ \mathit{FAIRSch},\ M_t.\mathit{init}) \sqsubseteq \mathrm{CI}(\mathtt{Load}\ p\, M_s,\ \mathit{FAIRSch},\ M_s.\mathit{init})$$

This theorem additionally allows each thread to have its precondition, P_{tid}, but can only be used for whole-program refinement, not for contextual refinement.

We conclude this section with the following remark. While the fairness logic contains only the minimal core rules, we are gathering confidence that it is powerful enough to handle various interesting examples: our flagship example, presented in the next section, involves non-trivial reasoning that spans multiple different notions of fairness. We believe further abstract constructs [5] could be derived on top of fairness logic and leave it as an interesting future work.

2.7.2 Example using the Fairness Logic

We demonstrate how the fairness logic is applied with a simplified but still illustrative example. The example focuses on *exploiting* fairness; for the case of validating fairness, we believe that the explanation in the previous section should be adequate for actual applications. This section assumes some familiarity with modern separation logic, *e.g.,* Iris [12], and uses the usual separation logic predicates with minimal explanations.

To show how to use the fairness logic in proofs relying on exploiting fairness, we inspect a simplified version of $\mathrm{CL_I}$, which assumes that memory access is atomic and removes the locks:

$$\curlyvee;\, X := 42;\, \curlyvee; \;\Big\|\; \texttt{do}\ \{\ \curlyvee;\ x := X;\ \curlyvee;\ \}\ \texttt{while}\ (x = 0)\ \curlyvee;\ \mathit{print}(x);\ \curlyvee; \tag{$\mathrm{SCL_I}$}$$

This program refines $\mathrm{CL_S}$, and the reasoning underlying the proof is similar to the one presented in §2.3.2: we construct a tuple that decreases throughout the program execution and perform induction on it. However, with the power of fairness logic, we can carry out the proof in a thread-local fashion—the most interesting proof obligation now becomes constructing a shared invariant that captures the rely-guarantee reasoning.

To begin with, we present an invariant one would set up, but without the fairness assertions:

$$(X \mapsto 0) \vee (X \mapsto 42 * \mathrm{Ex})$$

where $\mapsto$ is the usual points-to and Ex is a token for exclusive ownership in separation logic. This invariant states that the value stored in the memory location X is either 0 or 42, and in the latter case it also holds the token Ex, which means that thread 1 has written to X: thread 1 starts with Ex and hands it over to the invariant right after the write to X. Unfortunately, this invariant is too weak for the proof since it does not capture scheduler fairness; for the `while` loop in thread 2 to always terminate, it should be guaranteed that thread 1 will eventually write 42 to X, which in turn relies on scheduler fairness.

This intuition is precisely captured with a fairness assertion $\blacklozenge_1(\texttt{th}_1, n)$, representing that the fairness counter for thread 1 ($\texttt{th}_1$) is at most n. Then a correct invariant would be:

$$(X \mapsto 0 \ast \exists n.\, (\blacklozenge_1(\texttt{th}_1,\, n) \ast {=}(n))) \vee (X \mapsto 42 \ast \mathrm{Ex}) \qquad \text{(INV)}$$

where $=(n)$, together with $\leq(m)$, are predicates for monotonicity satisfying the following rules:

$$=(n) \vdash =(n) \ast \leq(n), \quad =(n) \ast \leq(m) \vdash (n \leq m), \quad =(m) \ast (n \leq m) \vdash \dot{\models\!\!\!\Rrightarrow} =(n)$$

Intuitively, $=(n)$ denotes the exact value of n, which can monotonically decrease, and $\leq(m)$ denotes that m is an upper bound of n [31]. Then INV states that the value at X is either 0 or 42, and when it is 0, there is a value n that represents how many times thread 1 can be starved by the scheduler ($\blacklozenge_1(\texttt{th}_1,\, n)$) and monotonically decreases ($=(n)$).

With INV, we demonstrate the core of the proof that $\mathrm{SCL_I}$ refines $\mathrm{CL_S}$—that the while-loop in thread 2 always terminates. Let us focus on the following:

$$\texttt{do}\ \{\ \curlyvee;\, x \coloneqq X;\ \curlyvee;\ \}\ \texttt{while}\ (x = 0)$$

For this piece of code, we stutter the source with the first $\curlyvee$ during simulation. Then at $x \coloneqq X;$, we destruct INV and get two cases: (*i*) $X \mapsto 42$ case terminates the loop by making the loop condition false, leaving (*ii*) $X \mapsto 0$ case, with a variable n, $\blacklozenge_1(\texttt{th}_1,\, n)$, and $=(n)$. In this case, we perform **(strong) induction on** n to conclude the proof, and we first obtain $\leq(n)$ from $=(n)$. Next, we proceed to execute $\curlyvee$ using YIELD-TGT and obtain $\lozenge_{\texttt{th}}$—note that thread 2 owns $\blacklozenge(\texttt{th}_2)$ from the start, and we can easily show the $X \mapsto 0$ case to guarantee the invariant INV. Then the loop iterates, almost bringing us to the state satisfying the induction hypothesis (after a trivial application of YIELD-TGT).

This time, we have $\leq(n)$ and $\lozenge_{\texttt{th}}$ along with INV, allowing us to finish the induction: we can obtain n' which satisfies $n' < n$ and the induction hypothesis. After destructing the invariant, we again inspect the $X \mapsto 0$ case, where it gives a fresh variable m, $\blacklozenge_1(\texttt{th}_1,\, m)$, and $=(m)$. First, $=(m)$ and $\leq(n)$ gives $m \leq n$

and case analysis leaves us with $m = n$ case since $m < n$ case concludes the induction. Next, substituting m with n, we apply DEC to $\blacklozenge_1(\mathtt{th}_1, n)$ and $\lozenge_{\mathtt{th}}$, obtaining $\blacklozenge_1(\mathtt{th}_1, n')$ where $n' < n$. Finally, after updating $=(n)$ to $=(n')$, we can conclude the induction since $n' < n$.

This line of reasoning is seamlessly extended to a tuple instead of a single value. To illustrate this, consider the following modification:

$$\curlyvee;\ \mathtt{skip};\ \curlyvee;\ X := 42;\ \curlyvee; \quad \left\Vert \quad \begin{array}{l} \mathtt{do}\ \{\ \curlyvee;\ x := X;\ \curlyvee;\ \}\ \mathtt{while}\ (x = 0)\ \curlyvee; \\ \mathit{print}(x);\ \curlyvee; \end{array} \right. \qquad (\text{SCL'}_{\text{I}})$$

Because of the inserted `skip`, we now need to incorporate the number of remaining $\curlyvee$s in thread 1, denoted l, into the induction (as we did in §2.3.2). To encode l in the invariant, we use the usual authoritative assertions $\bullet(l)$ and $\circ(l)$, satisfying the following rules [11]:

$$\bullet(a) \ast \circ(b) \vdash (a = b), \qquad \bullet(a) \ast \circ(b) \vdash \dot{\Rrightarrow} \bullet(c) \ast \circ(c)$$

Then an invariant for proving that SCL'$_{\text{I}}$ refines CL$_{\text{S}}$ is:

$$(X \mapsto 0 \ast \exists n, l.\, (\blacklozenge_1(\mathtt{th}_1, n) \ast \bullet(l) \ast =(l, n))) \vee (X \mapsto 42 \ast \text{Ex}) \qquad (\text{INV'})$$

The other half, $\circ(l)$, is owned by thread 1 and tracks the number of remaining $\curlyvee$s in thread 1. Using INV', we can prove the goal with induction; we omit the details, which is similar to the one above.

2.8 Case Study

Composing all of our results, we can prove the motivating example: CL$_{\text{I}}$ using TicketLock$_{\mathtt{FWMM}}$ and under `FIFOSch` refines CL$_{\text{S}}$ under `FAIRSch`. To state this, we first wrap up each as a module, $\mathtt{CL}_{\mathtt{I},\mathtt{tk}}$ and $\mathtt{CL}_{\mathtt{S}}$, which contains appropriate functions and an initial state. Then we load the modules with a configuration p that maps threads 1 and 2 to corresponding functions, and state the desired refinement, which we prove by assembling the refinement results by the transitivity of refinement:

$$\begin{aligned}
&\mathrm{CI}(\texttt{Load}\,p\,\texttt{CL}_{\texttt{I},\texttt{tk}},\ \texttt{FIFOSch},\ \texttt{CL}_{\texttt{I},\texttt{tk}}.\texttt{init})\\
&\sqsubseteq \mathrm{CI}(\texttt{Load}\,p\,\texttt{CL}_{\texttt{I},\texttt{tk}},\ \texttt{FAIRSch},\ \texttt{CL}_{\texttt{I},\texttt{tk}}.\texttt{init})\\
&\sqsubseteq \mathrm{CI}(\texttt{Load}\,p\,\texttt{CL}_{\texttt{I},\texttt{abs}},\ \texttt{FAIRSch},\ \texttt{CL}_{\texttt{I},\texttt{abs}}.\texttt{init})\\
&\sqsubseteq \mathrm{CI}(\texttt{Load}\,p\,\texttt{CL}_{\texttt{S}},\ \texttt{FAIRSch},\ \texttt{CL}_{\texttt{S}}.\texttt{init})
\end{aligned}$$

where the first and second refinements are direct applications of Theorems 2.5.2 and 2.6.2. Theorem 2.6.2 and the final refinement are proved using fairness logic (Theorems 2.7.1 and 2.7.2). Also, we remark that client-library modular reasoning manifests itself in the application of Theorem 2.6.2 at the second refinement. This result, including the whole theory of FOS, is fully mechanized in Coq.

2.9 Related Work and Discussion

Various concepts of fairness have been studied within the literature, including scheduler fairness [17, 20], progress properties for concurrent objects [9, 4, 23], weak memory models [15], and model checking [1].

As shown throughout the paper, the definition of fairness described in Definition 2.2.1 is general enough to capture all of these concepts of fairness, which in turn makes these concepts expressible in FOS. In this section, we describe some pieces of previous work in more detail and compare the advantages (and disadvantages) that FOS provides as a framework compared to existing work.

Progress Properties of Concurrent Programs. An interesting line of work in establishing progress properties of a concurrent program is to prove that a program refines some operational specification that encodes the desired progress property [7]. A prime example is LiLi [22, 23], which provides a method to express such specs, then allows users to prove that a program contextually refines the spec (thereby ensuring that a progress property holds).

However, the specs that LiLi provides do not express fairness directly, and instead employ an explicit queue of definite actions within the semantics to capture the idea of fairness. LiLi also does not provide any machinery for verifying clients that rely on fairness assumptions outside of scheduler fairness—and thus cannot exploit fairness, as we have done in FOS.

[24] studies relations between all common progress properties and operational definitions, based on contextual refinements. They prove that each progress property is equivalent to some specific type of contextual refinement for linearizable objects. We believe that this approach can be applied to our theory, such that proving contextual refinements can formally establish desired progress properties, which we leave for future work.

Liveness and Temporal Logics. Fairness shares many similarities with *liveness*, which states that a 'good thing' must eventually happen at some point during the execution of a program: for example, that a process must 'get scheduled' eventually. Such liveness properties are typically encoded via temporal logics [26]—most prominently, linear temporal logic (LTL) [14]—which state properties that an entire trace of a program execution must satisfy.

We note that while fairness and liveness are similar concepts with many overlapping applications, the definition of fairness presented in this paper does have differences with the standard notion of liveness: liveness requires that a `good` event *must* happen, while with fairness, it is fine that a `good` event does not occur as long as each fairness id only accumulates a finite number of `bad` events.

Concurrent Separation Logic. Concurrrent separation logics [25, 2] extend modern separation logics with the goal of proving safety properties of concurrent programs. One notable feature of concurrent separation logics is that they focus on thread-local and compositional reasoning, in order to reduce proof burdens.

One notable work in this area is TaDA-Live [5], which extends [27], a concurrent separation logic for verifying total correctness of client programs using concurrent objects. TaDA-Live provides fairness-aware specifications of concurrent objects in the style of Hoare triples, and exploits scheduler fairness during verification, with a focus on proving that a set of threads terminate under scheduler fairness. Such termination proofs often rely on induction, where one must identify a decreasing value to perform induction upon (such as the tuple of values from §2.3). While abstracted away in this paper, identifying and constructing these values presents the main challenge in FOS proofs; a major contribution of TaDA-Live is that it provides a high degree of abstraction to hide this complexity, providing users with a simple proof interface.

However, TaDA-Live cannot provide any guarantees for nonterminating programs: for example, TaDA-Live cannot be used to express progress properties for programs that loop indefinitely. TaDA-Live is also only capable of unary reasoning and cannot be used for proving refinement.

There has been work on proving termination-preserving refinement under a fair scheduler using relational separation logic [29, 6]. However, such approaches do not support reasoning about fairness directly, or exploiting fairness.

Bibliography

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[2] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

[3] M. Cho, S.-H. Lee, D. Lee, C.-K. Hur, and O. Lahav. Sequential reasoning for optimizing compilers under weak memory concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 213–228, New York, NY, USA, 2022. Association for Computing Machinery.

[4] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, oct 1971.

[5] E. D'Osualdo, J. Sutherland, A. Farzan, and P. Gardner. Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, 43(4), nov 2021.

[6] L. Gäher, M. Sammler, S. Spies, R. Jung, H.-H. Dang, R. Krebbers, J. Kang, and D. Dreyer. Simuliris: A separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[7] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *Proceedings of the 38th International Conference on Automata, Languages*

and Programming - Volume Part II, ICALP'11, page 453–465, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, 2015.

[9] M. Herlihy and N. Shavit. On the nature of progress. In A. Fernàndez Anta, G. Lipari, and M. Roy, editors, *Principles of Distributed Systems*, pages 313–328, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[10] G. Hessenberg. *Grundbegriffe der mengenlehre*, volume 1. Vandenhoeck & Ruprecht, 1906.

[11] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[12] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices*, 50(1):637–650, 2015.

[13] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 175–189, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Y. Kesten, A. Pnueli, and L.-o. Raviv. Algorithmic verification of linear temporal logic specifications. In *Automata, Languages and Programming: 25th International Colloquium, ICALP'98 Aalborg, Denmark, July 13–17, 1998 Proceedings 25*, pages 1–16. Springer, 1998.

[15] O. Lahav, E. Namakonov, J. Oberhauser, A. Podkopaev, and V. Vafeiadis. Making weak memory models fair. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[16] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. Repairing sequential consistency in c/c++ 11. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 618–632. ACM, 2017.

[17] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[18] D. Lee, M. Cho, J. Kim, S. Moon, Y. Song, and C.-K. Hur. Fair operational semantics. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[19] S.-H. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C.-K. Hur, O. Lahav, and V. Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 362–376, New York, NY, USA, 2020. Association for Computing Machinery.

[20] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In S. Even and O. Kariv, editors, *Automata, Languages and Programming*, pages 264–277, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[21] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2006, 2006.

[22] H. Liang and X. Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page

385–399, New York, NY, USA, 2016. Association for Computing Machinery.

[23] H. Liang and X. Feng. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[24] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *Proceedings of the 24th International Conference on Concurrency Theory*, CONCUR'13, page 227–241, Berlin, Heidelberg, 2013. Springer-Verlag.

[25] P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1–3):271–307, apr 2007.

[26] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, jul 1982.

[27] P. Rocha Pinto, T. Dinsdale-Young, P. Gardner, and J. Sutherland. Modular termination verification for non-blocking concurrency. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, page 176–201, Berlin, Heidelberg, 2016. Springer-Verlag.

[28] Y. Song, M. Cho, D. Lee, C.-K. Hur, M. Sammler, and D. Dreyer. Conditional contextual refinement. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[29] J. Tassarotti, R. Jung, and R. Harper. A higher-order logic for concurrent termination-preserving refinement. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*, page 909–936, Berlin, Heidelberg, 2017. Springer-Verlag.

[30] The Coq Development Team. The Coq proof assistant 8.13.2 reference manual, 2021. `https://coq.github.io/doc/V8.13.2/refman/`.

[31] A. Timany and L. Birkedal. Reasoning about monotonicity in separation logic. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 91–104, New York, NY, USA, 2021. Association for Computing Machinery.

[32] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.

초록

본 논문에서는 프로그램의 공정성(fairness) 성질들을 기술하고 프로그램 검증에 이용할 수 있도록 하는 것을 목적으로 하는 이론인 FOS를 제시한다. FOS는 Fair Operational Semantics를 의미하고, 프로그램의 공정성 성질들을 실행 의미를 통해 표현하고 이용할 수 있도록 하는 이론이다. 공정성 성질들은 프로그램의 실행에 있어서 나쁜 이벤트가 좋은 이벤트 없이 무한히 발생하지 않는다는 것을 나타낸다. 이러한 공정성 성질들은 프로그램 검증에 있어 두 가지 중요성을 갖고 있다. 먼저, 공정성 성질들은 검증하고자 하는 프로그램에 대해 안정성 성질보다 더욱 정확한 스펙으로서 사용될 수 있다. 다음으로, 동시성 프로그램의 종료 성질 검증 등의 프로그램 검증은 많은 경우 공정성 성질들에 의존한다. 본 논문에서는 FOS가 이 두 가지 경우 모두에 잘 사용될 수 있다는 것을 보인다. 구체적으로, FOS를 이용하여 다양한 공정성 성질들을 표현하는 스펙을 개발하고, 각 쓰레드를 개별로 고려해도 되는 증명 기법을 개발하고, 또한 이 기법을 이용하여 다양한 예시들을 검증한다.

주요어: 실행 의미, 동시성 프로그래밍, 공정성 성질, 프로그램 검증

학번: 2021-26995

Acknowledgements

이 논문은 여러 사람의 도움으로 완성되었습니다. 먼저, 제 지도교수님이신 허충길 교수님께 감사드립니다. 연구에 있어 교수님의 통찰력과 열정, 그리고 본질을 추구하시는 모습에서 많은 것을 배웠습니다. 다음으로, 연구실 동료들, 특히 조민기 박사, 이성환 박사, 송용주 박사에게 감사드립니다. 연구에 필요한 아이디어와 기술, 그리고 연구자에게 필요한 소통 능력 등 많은 것을 배웠습니다. 마지막으로, 저를 응원하고 지탱해 준 가족, 부모님과 동생에게 감사합니다.