Encoding fairness in a synchronous concurrent program algebra[⋆]

Ian J. Hayes and Larissa A. Meinicke

The University of Queensland, Brisbane, Queensland, Australia

Abstract. Concurrent program refinement algebra provides a suitable basis for supporting mechanised reasoning about shared-memory concurrent programs in a compositional manner, for example, it supports the rely/guarantee approach of Jones. The algebra makes use of a synchronous parallel operator motivated by Aczel's trace model of concurrency and with similarities to Milner's SCCS. This paper looks at defining a form of fairness within the program algebra. The encoding allows one to reason about the fair execution of a single process in isolation as well as define fair-parallel in terms of a base parallel operator, of which no fairness properties are assumed. An algebraic theory to support fairness and fair-parallel is developed.

1 Introduction

In shared memory concurrency, standard approaches to handling fairness [16,13] focus on defining a fair parallel operator, $c \parallel_f d$, that ensures each process gets its fair share of processor cycles. That complicates reasoning about a single process running as part of a parallel composition because its progress is determined in part by the fair parallel operator. In this paper we first focus on a single process that is run fairly with respect to its environment. That allows one to reason about its progress properties in relative isolation, although one does need to rely on its environment (i.e. all processes running in parallel with it) satisfying assumptions the single process makes about its environment. Fair parallel composition of processes can then be formulated as (unfair) parallel composition of fair executions of each of the individual processes (i.e. fair-execution(c) $\parallel$ fair-execution(d)), where fair-execution of a command is defined below.

Unfair parallel. For a parallel composition, $c \parallel d$, the execution of c may be pre-empted forever by the execution of d, or vice versa. For example, execution of

$$x := 1 \parallel \mathbf{do}\, x \neq 1 \rightarrow y := y + 1\ \mathbf{od} \tag{1}$$

with x initially zero may not terminate if the right side loop pre-empts the left side assignment forever [17]. A minimal fairness assumption is that neither process of a parallel composition can be pre-empted by the other process indefinitely.

[⋆] This work was supported by Australian Research Council (ARC) Discovery Project DP130102901.

Aczel traces. The denotational semantics that we use for concurrency [3] is based on Aczel's model [2,4,5], in which the possible behaviours of a process, specified by *Aczel traces*, describe both the steps taken by the process itself as well as the steps taken by its environment. An Aczel trace is a sequence of atomic steps from one state σ to the next σ', in which each atomic step is either a *program step* of the form $\Pi(\sigma, \sigma')$ or an *environment step* of the form $\mathcal{E}(\sigma, \sigma')$. Parallel composition has an interleaving interpretation and so program and environment steps are disjoint. Infinite atomic-step sequences denote non-terminating executions, and finite sequences are labeled to differentiate those that (i) *terminate*, (ii) *abort* or (iii) become *infeasible* after the last atomic step in the sequence. Abortion represents failure (e.g. failure caused by a violation of environment assumptions), that may be *refined* (i.e. implemented) by any subsequent behaviour. Infeasibility may arise due to conflicting constraints in specifications, and is a refinement of any subsequent behaviours. Because each Aczel trace of a process defines both its behaviour as well as the behaviour of its environment, it is possible to include assumptions and constraints (including fairness) on the environment of a process in its denotation – the set of Aczel traces that it (or any valid implementation of it) may perform.

When two processes are combined in parallel, each must respect the environmental constraints placed upon it by the other process – unless either fails, in which case the parallel composition also fails. For example, assuming neither process has failed, one process may only take a program step $\Pi(\sigma, \sigma')$ if its parallel process may perform a step $\mathcal{E}(\sigma, \sigma')$, which permits its environment to take that program step at that point of execution. This is achieved by requiring parallel processes to synchronise on every atomic step they take: a program step $\Pi(\sigma, \sigma')$ of one process matches the corresponding environment step $\mathcal{E}(\sigma, \sigma')$ of the other to give a program step $\Pi(\sigma, \sigma')$ of the parallel composition, and identical environment steps of both processes match to give that environment step for the parallel composition. Attempting to synchronise other steps is infeasible.

Let π specify the nondeterministic command that executes a single atomic program step and then terminates, but does not constrain the state-transition made by that step, that is, π could take $\Pi(\sigma, \sigma')$ for any possible states σ and σ'. Similarly, let ϵ represent the non-deterministic command that executes any single atomic environment step and then terminates [3,8,9]. Neither π nor ϵ is allowed to fail: they do not contain aborting behaviour.

The command $c^\star$ represents finite iteration of command c, zero or more times, and c^ω represents finite or infinite iteration of c, zero of more times. The command c^∞ is the infinite iteration of c. Note that c^ω splits into finite and infinite iteration of c, where $\sqcap$ represents (demonic) nondeterministic choice.

$$c^\omega = c^\star \sqcap c^\infty \tag{2}$$

Imposing fairness. If a process is pre-empted forever its behaviour becomes an infinite execution of any environment steps, i.e. ϵ^∞. The process **fair** that allows any behaviour, except abortion and pre-emption by its environment forever, can be defined by

$$\mathbf{fair} \mathrel{\widehat{=}} \epsilon^\star \; (\pi \; \epsilon^\star)^\omega \tag{3}$$

where juxtaposition represents sequential composition. The process $\mathbf{fair}$ requires all contiguous subsequences of environment steps to be finite. A process representing *fair execution* of a process c is represented by

$$c \Cap \mathbf{fair}$$

where the *weak conjunction*, $c \Cap d$, of c and d behaves as both c and d unless one of them aborts, in which case $c \Cap d$ aborts [6,3]. Because $\mathbf{fair}$ never aborts, any aborting behaviour of $c \Cap \mathbf{fair}$ arises solely from c. In this way, c is constrained to be fair until it fails, if ever. Weak conjunction is associative, commutative and idempotent; it has identity $\mathbf{chaos}$ defined in terms of iteration of any number of atomic steps, where α represents a single atomic step, either program or environment.

$$\alpha = \pi \sqcap \epsilon \tag{4}$$

$$\mathbf{chaos} \mathrel{\widehat{=}} \alpha^{\omega} \tag{5}$$

Because program and environment steps are disjoint, the conjunction of these commands is the infeasible command $\top$, i.e. $\pi \Cap \epsilon = \top$.

Our interpretation of the execution of the process,

$$\mathbf{do}\,\mathsf{true} \rightarrow y := y + 1\ \mathbf{od}\ , \tag{6}$$

from an initial state in which y is zero allows the loop to be pre-empted forever by its environment and thus does not guarantee that y is ever set to, say, 7. In contrast, the fair execution of (6),

$$\mathbf{do}\,\mathsf{true} \rightarrow y := y + 1\ \mathbf{od} \Cap \mathbf{fair}\ , \tag{7}$$

rules out pre-emption by its environment forever and hence ensures that eventually y becomes 7 (or any other natural number).

Fair termination. The command $\mathbf{term}$ allows only a finite number of program steps but does not rule out infinite pre-emption by its environment. It is defined as follows [6,3], recalling that $\alpha = \pi \sqcap \epsilon$.

$$\mathbf{term} \mathrel{\widehat{=}} \alpha^{\star}\ \epsilon^{\omega} \tag{8}$$

If $\mathbf{term}$ is combined with $\mathbf{fair}$, pre-emption by the environment forever is eliminated giving a stronger termination property that allows only a finite number of both program and environment steps, see Lemma 14 (term-fair).

$$\mathbf{term} \Cap \mathbf{fair} = \alpha^{\star}$$

The notation $c \sqsubseteq d$ means c is refined (or implemented) by d and is defined by,

$$c \sqsubseteq d \mathrel{\widehat{=}} ((c \sqcap d) = c)\ . \tag{9}$$

Hence if $\mathbf{term} \sqsubseteq c$, then $\mathbf{term} \Cap \mathbf{fair} \sqsubseteq c \Cap \mathbf{fair}$, i.e. fair execution of c gives strong termination, meaning that there are only a finite number of steps overall, both program and environment.

Fairness and concurrency. Consider the following variation of example (1).

$$((x := 1) \Cap \mathbf{fair}) \parallel (\mathbf{do}\, x \neq 1 \rightarrow y := y + 1 \,\mathbf{od} \Cap \mathbf{fair}) \tag{10}$$

The fair execution of $x := 1$ rules out infinite pre-emption by the right side and hence x is eventually set to one, and hence the right side also terminates thus ensuring termination of the parallel composition. Note that

$$(c \parallel d) \Cap \mathbf{fair} \sqsubseteq (c \Cap \mathbf{fair}) \parallel (d \Cap \mathbf{fair})$$

but the reverse refinement does not hold in general because $(c \parallel d) \Cap \mathbf{fair}$ does not rule out c being pre-empted forever by d (or vice versa) within the parallel; it only rules out the whole of the parallel composition from being preempted by its environment forever.

Parallel with synchronised termination. The parallel operator $\parallel$ is interpreted as synchronous parallel for which every step of the parallel (until failure of either process) must be a synchronisation of steps of its component processes: a program and environment step synchronise to give a program step, $\pi \parallel \epsilon = \pi$, two environment steps synchronise to give an environment step, $\epsilon \parallel \epsilon = \epsilon$ and both the processes must terminate together, $\mathbf{nil} \parallel \mathbf{nil} = \mathbf{nil}$. This is in contrast to the *early-termination* interpretation of parallel in which, if one process terminates the parallel composition reduces to the other process. The command ϵ^ω, referred to as $\mathbf{skip}$,

$$\mathbf{skip} \mathrel{\widehat{=}} \epsilon^\omega \tag{11}$$

is the identity of parallel composition, meaning that it permits any possible environment behaviour when executed in parallel with any other command, e.g. $c \parallel \mathbf{skip} = c$. A command c for which

$$c = c\,\mathbf{skip} \tag{12}$$

is said to be *unconstrained after program termination*. When it is executed in parallel with another command, then after termination of c, the parallel composition $c \parallel d$ does reduce to the other command, d. If d is also unconstrained after program termination, then $c \parallel d$ corresponds to the early-termination interpretation of parallel. Moreover, $c \parallel d$ is then also unconstrained after program termination, e.g. $c \parallel d = (c \parallel d)\,\mathbf{skip}$, see Lemma 8 (par-skip). In this way (12) can be perceived as a healthiness condition, that is preserved by parallel composition of healthy commands.

The fair execution of any process c constrains the environment, even after the termination of the program steps in c, so that it cannot execute an infinite number of steps in a row, e.g. $\mathbf{term} \Cap \mathbf{fair} = \alpha^\star$. This means that it is not healthy (12), and so for parallel with synchronised termination, simply conjoining $\mathbf{fair}$ to both sides of a synchronous parallel can lead to infeasibility. Consider another of Van Glabbeek's examples [17]:

$$(x := 1 \Cap \mathbf{fair}) \parallel (\mathbf{do}\, \mathsf{true} \rightarrow y := y + 1 \,\mathbf{od} \Cap \mathbf{fair})\ . \tag{13}$$

The fair execution of $x := 1$ rules out infinite pre-emption by the right side loop, ensuring x is assigned one, but fair execution of $x := 1$ forces termination of the left side, including environment steps, which as the right side is non-terminating leads to an infeasible parallel composition. To remedy this one needs to allow infinite pre-emption of a branch in a fair parallel *once the command in the branch has terminated*. For a command c satisfying (12) we have that

$$(c \Cap \mathbf{fair})\, \mathbf{skip} \tag{14}$$

represents fair execution of c *until program termination*. Like the original command c, it remains unconstrained after program termination (i.e. healthy). For the example above, we have implicitly that $x := 1$ and the loop ($\mathbf{do}\, \mathsf{true} \to y := y + 1\, \mathbf{od}$) are unconstrained after program termination, and so only requiring both branches to execute fairly until program termination we get

$$(x := 1 \Cap \mathbf{fair})\, \mathbf{skip} \parallel (\mathbf{do}\, \mathsf{true} \to y := y + 1\, \mathbf{od} \Cap \mathbf{fair})\, \mathbf{skip} \tag{15}$$

which is no longer infeasible, since the second process is allowed to execute forever after termination of the program steps in the first.

That leads to the following definition for fair parallel,

$$c \underset{f}{\parallel} d \mathrel{\widehat{=}} (c \Cap \mathbf{fair})\, \mathbf{skip} \parallel (d \Cap \mathbf{fair})\, \mathbf{skip} \tag{16}$$

which imposes fairness on c until it terminates, and similarly for d.

Our theory of fairness is based on the synchronous concurrent refinement algebra, which is summarised in Sect. 2 and Sect. 3 gives a set of lemmas about iterations in the algebra. Sect. 4 gives basic properties of the command **fair**, while Sect. 5 gives properties of **fair** combined with (unfair) concurrency and Sect. 6 uses these to derive properties of the fair-parallel operator which is defined in terms of (unfair) parallel (16).

2 Synchronous concurrent refinement algebra

The synchronous concurrent refinement algebra is defined in [8,9]. In this section we introduce the aspects that are used to define and reason about fairness in this paper. A model for the algebra based on Aczel traces, as discussed in the introduction, can be found in [3].

A concurrent refinement algebra with atomic steps ($\mathcal{A}$), and synchronisation operators parallel ($\parallel$) and weak conjunction ($\Cap$) is a two-sorted algebra

$$(\mathcal{C}, \mathcal{A}, \bigsqcap, \bigsqcup, ;, \parallel, \Cap, !\,, \mathbf{nil}, \alpha, \mathbf{skip}, \mathbf{chaos}, \epsilon)$$

where the carrier set $\mathcal{C}$ is interpreted as the set of *commands* and forms a complete distributive lattice with meet ($\bigsqcap$), referred to as *choice*, and join ($\bigsqcup$), referred to as *conjunction*, and refinement ordering given by (9), where we use $c \sqcap d \mathrel{\widehat{=}} \bigsqcap\{c, d\}$, and $c \sqcup d \mathrel{\widehat{=}} \bigsqcup\{c, d\}$ to represent the meet and join over pairs of elements. The least and greatest elements in the lattice are the aborting command $\bot \mathrel{\widehat{=}} \bigsqcap \mathcal{C}$, and the infeasible command $\top \mathrel{\widehat{=}} \bigsqcup \mathcal{C}$, respectively. The binary operator ";", with identity element $\mathbf{nil}$, represents *sequential* composition (and satisfies the axioms listed in Fig. 1), however we abbreviate $c ; d$ to $c\ d$ throughout this paper.

For $i \in \mathbb{N}$, we use c^i to represent the fixed-iteration of the command c, i times. It is inductively defined by $c^0 \mathrel{\widehat{=}} \mathbf{nil}$, $c^{i+1} \mathrel{\widehat{=}} c\, c^i$. More generally, fixed-point operators finite iteration (*), finite or infinite iteration ($^\omega$), and infinite iteration ($^\infty$) are defined using the least (μ) and greatest (ν) fixed-point operators of the complete distributive lattice of commands,

$$c^\star \mathrel{\widehat{=}} (\nu x.\mathbf{nil} \sqcap c\, x) \qquad (17) \qquad\qquad c^\infty \mathrel{\widehat{=}} c^\omega\, \top \qquad (19)$$

$$c^\omega \mathrel{\widehat{=}} (\mu x.\mathbf{nil} \sqcap c\, x) \qquad (18)$$

and satisfy the properties outlined in Sect. 3.

The second carrier set $\mathcal{A} \subseteq \mathcal{C}$ is a sub-algebra of *atomic step commands*, defined so that $(\mathcal{A}, \sqcap, \sqcup, !\,, \top, \alpha)$ forms a Boolean algebra with greatest element $\top$ (also the greatest command), which can be thought of the atomic step that is disabled from all initial states, the least element α, the command that can perform any possible atomic step. The negation of an atomic step $a \in \mathcal{A}$, written $!\,a$, represents all of the atomic steps that are not in a. Distinguished atomic step $\epsilon \in \mathcal{A}$ is used to stand for any possible environment step, and its complement, $\pi \mathrel{\widehat{=}} !\,\epsilon$, is then the set of all possible program steps, giving us that $\alpha = \pi \sqcap \epsilon$.

Both *parallel* composition ($\parallel$) and *weak conjunction* ($\Cap$) are instances of the synchronisation operator ($\otimes$), in which parallel has command identity $\mathbf{skip} = \epsilon^\omega$, and atomic-step identity ϵ; and weak conjunction has command identity $\mathbf{chaos} = \alpha^\omega$, and atomic-step identity α. As well as satisfying the synchronisation axioms from Fig. 1, a number of additional axioms, also listed in the figure, are assumed. These include, for example, that both operators are abort-strict, (36) and (37), weak conjunction is idempotent (38), and they include assumptions about the synchronisation of atomic steps, e.g. (39) and (40).

We follow the convention that c and d stands for arbitrary commands, and a and b for atomic step commands. Further, subscripted versions of these stand for entities of the same kind. We also assume that choice ($\sqcap$) has the lowest precedence, and sequential composition has the highest; and we use parentheses to disambiguate other cases.

3 Properties of iterations

In this section we outline the iteration properties required in this paper. Omitted or abbreviated proofs can be found in [7].

First, from [8,9], we have that the iteration operators satisfy the basic properties listed in Fig. 2. The following lemma (also from [8]), captures that prefixes of finite iterations of atomic steps $a^\star\ c$ and $b^\star\ d$ combine in parallel until either $a^\star$ or $b^\star$ or both complete. If both $a^\star$ and $b^\star$ complete together, the remaining commands after the prefixes run in parallel: $c \parallel d$. If the first completes before the second, c runs in parallel with at least one b followed by d, and symmetrically if the second completes before the first.

Lemma 1 (finite-finite-prefix).

$$a^\star\ c \parallel b^\star\ d = (a \parallel b)^\star\ ((c \parallel d) \sqcap (c \parallel b\, b^\star\ d) \sqcap (a\, a^\star\ c \parallel d))$$

Sequential

$$c_0\;(c_1\;c_2) = (c_0\;c_1)\;c_2 \quad (20)$$
$$c\;\mathbf{nil} \;=\; c \;=\; \mathbf{nil}\;c \quad (21)$$
$$\bot\;c = \bot \quad (22)$$
$$(\bigsqcap C)\;d = \bigsqcap_{c\in C}(c\;d) \quad (23)$$
$$D \neq \emptyset \;\Rightarrow\; c\;(\bigsqcap D) = \bigsqcap_{d\in D}(c\;d) \quad (24)$$

Synchronisation operators parallel and weak conjunction Both parallel ($\|$) and weak conjunction ($\Cap$) are instances of the synchronisation operator ($\otimes$). For parallel we take the identity command $\mathcal{I}d$ to be $\mathbf{skip}$, and atomic-step identity $\mathbf{1}$ to be ϵ, and for weak conjunction we take $\mathcal{I}d$ to be $\mathbf{chaos}$ and $\mathbf{1}$ to be α.

$$c_0 \otimes (c_1 \otimes c_2) = (c_0 \otimes c_1) \otimes c_2 \quad (25)$$
$$c \otimes d = d \otimes c \quad (26)$$
$$c \otimes \mathcal{I}d = c \quad (27)$$
$$D \neq \emptyset \;\Rightarrow\; c \otimes (\bigsqcap D) = \bigsqcap_{d\in D}(c \otimes d) \quad (28)$$
$$a \otimes \mathbf{1} = a \quad (29)$$
$$\mathbf{nil} \otimes \mathbf{nil} = \mathbf{nil} \quad (30)$$
$$\mathbf{nil} \otimes a\;c = \top \quad (31)$$
$$a \otimes b \in \mathcal{A} \quad (32)$$
$$(a\;c) \otimes (b\;d) = (a \otimes b)\;(c \otimes d) \quad (33)$$
$$a^{\infty} \otimes b^{\infty} = (a \otimes b)^{\infty} \quad (34)$$
$$(c_0\;d_0) \otimes (c_1\;d_1) \sqsubseteq (c_0 \otimes c_1)\;(d_0 \otimes d_1) \quad (35)$$

Additional parallel and weak conjunction axioms As well as satisfying the synchronisation axioms the following axioms of parallel and weak conjunction are assumed to hold.

$$c \parallel \bot = \bot \quad (36)$$
$$c \Cap \bot = \bot \quad (37)$$
$$c \Cap c = c \quad (38)$$
$$\pi \parallel \pi = \top \quad (39)$$
$$\pi \Cap \epsilon = \top \quad (40)$$
$$c \Cap \alpha^{i} = c \parallel \epsilon^{i} \quad (41)$$
$$c \Cap \alpha^{\infty} = c \parallel \epsilon^{\infty} \quad (42)$$
$$(c_0 \Cap \alpha^{i})\;d_0 \parallel (c_1 \Cap \alpha^{i})\;d_1 = ((c_0 \Cap \alpha^{i}) \parallel (c_1 \Cap \alpha^{i}))\;(d_0 \parallel d_1) \quad (43)$$
$$(c_0 \Cap \alpha^{i})\;d_0 \Cap (c_1 \Cap \alpha^{i})\;d_1 = (c_0 \Cap c_1 \Cap \alpha^{i})\;(d_0 \Cap d_1) \quad (44)$$
$$(c_0 \parallel d_0) \Cap (c_1 \parallel d_1) \sqsubseteq (c_0 \Cap c_1) \parallel (d_0 \Cap d_1) \quad (45)$$

Fig. 1. Axioms for the synchronous concurrent refinement algebra. We let $c, d \in \mathcal{C}$ be commands, $C, D \in \mathbb{P}\,\mathcal{C}$ be sets of commands, $a, b \in \mathcal{A}$ be atomic steps, and $i \in \mathbb{N}$ be a natural number.

$$c^{\star} = \mathbf{nil} \sqcap c\;c^{\star} \quad (46)$$
$$c^{\omega} = \mathbf{nil} \sqcap c\;c^{\omega} \quad (47)$$
$$c^{\omega} = c^{\star} \sqcap c^{\infty} \quad (48)$$
$$c^{\star} = \textstyle\bigsqcap_{i\in\mathbb{N}} c^{i} \quad (49)$$
$$d \sqcap c\;x \sqsubseteq x \;\Longrightarrow\; c^{\omega}\;d \sqsubseteq x \quad (50)$$
$$x \sqsubseteq d \sqcap c\;x \;\Longrightarrow\; x \sqsubseteq c^{\star}\;d \quad (51)$$
$$c\;(d\;c)^{\star} \;=\; (c\;d)^{\star}\;c \quad (52)$$
$$c\;(d\;c)^{\omega} \;=\; (c\;d)^{\omega}\;c \quad (53)$$
$$(c \sqcap d)^{\omega} \;=\; c^{\omega}\;(d\;c^{\omega})^{\omega} \quad (54)$$

Fig. 2. Basic properties of iteration operators for commands $c, d, x \in \mathcal{C}$.

The next lemma is similar to Lemma 1, except one of the prefixes is finite and the other is possibly infinite.

Lemma 2 (finite-omega-prefix).

$$a^\star\ c \parallel b^\omega\ d = (a \parallel b)^\star\ ((c \parallel d) \sqcap (c \parallel b\, b^\omega\ d) \sqcap (a\, a^\star\ c \parallel d))$$

The following lemma uses the fact that program steps do not synchronise with other program steps in parallel (39), to simplify the parallel composition of two iterations.

Lemma 3 (iterate-pi-par-pi). $(\pi\ c)^\omega \parallel (\pi\ d)^\omega = \mathbf{nil}$

Proof. The proof uses (47), distribution and then (30), (31) twice, (33), and (39). □

Lemma 4 (iterate-pi-sync-atomic). *For either synchronisation operator,* $\parallel$ *or* $\Cap$*, and atomic step command* a*,*

$$(\pi\ c)^\omega \otimes a\ d = (\pi \otimes a)\ (c\ (\pi\ c)^\omega \otimes d)\ .$$

Proof. The proof uses (47), distribution and then (31) and (33). □

Lemma 5 (distribute-infeasible-suffix). *For any synchronisation operator* $(\otimes)$ *that is abort strict, i.e.* $(c \otimes \bot) = \bot$ *for all* c*, then we have that for any commands* c*,* d*,*

$$c \otimes d\ \top = (c \otimes d)\ \top\ .$$

Lemma 6 (infinite-annihilates). $(c \Cap \alpha^\infty)\ d_1 = (c \Cap \alpha^\infty)\ d_2\ .$

Proof. The result follows straightforwardly from the fact that weak conjunction is abort strict (37), $\alpha^\infty = \alpha^\infty\ \top$ from (19) and Lemma 5 (distribute-infeasible-suffix), together with the fact that $\top\ d_1 = \top = \top\ d_2$ from (23) by taking C in (23) to be empty. □

Taking d_2 to be **nil** in the above lemma gives $(c \Cap \alpha^\infty)\ d = c \Cap \alpha^\infty$, for any d.

Lemma 7 (sync-termination). *For commands* c *and* d *such that* $c = c \Cap \alpha^\star$ *and* $d = d \Cap \alpha^\star$*,*

$$(c\ a^\star \parallel d\ b^\star)\ (a^\omega \parallel b^\omega) = c\ a^\omega \parallel d\ b^\omega$$

The following lemma gives us that parallel composition preserves the healthiness property (12).

Lemma 8 (par-skip). $(c\ \mathbf{skip} \parallel d\ \mathbf{skip})\ \mathbf{skip} = c\ \mathbf{skip} \parallel d\ \mathbf{skip}$

Proof. Refinement from left to right is straightforward because $\mathbf{skip} \sqsubseteq \mathbf{nil}$:

$$(c\ \mathbf{skip} \parallel d\ \mathbf{skip})\ \mathbf{skip} \sqsubseteq (c\ \mathbf{skip} \parallel d\ \mathbf{skip})\ \mathbf{nil} = c\ \mathbf{skip} \parallel d\ \mathbf{skip}\ .$$

Refinement from right to left can be shown as follows.

$$\begin{array}{ll}
 & c\ \mathbf{skip} \parallel d\ \mathbf{skip} \\
= & \quad \text{as } \mathbf{skip} = \mathbf{skip}\ \mathbf{skip} \\
 & c\ \mathbf{skip}\ \mathbf{skip} \parallel d\ \mathbf{skip}\ \mathbf{skip} \\
\sqsubseteq & \quad \text{by sync-interchange-seq (35)} \\
 & (c\ \mathbf{skip} \parallel d\ \mathbf{skip})\ (\mathbf{skip} \parallel \mathbf{skip}) \\
= & \quad \mathbf{skip} \text{ is the identity of parallel composition} \\
 & (c\ \mathbf{skip} \parallel d\ \mathbf{skip})\ \mathbf{skip}
\end{array}$$

□

4 Properties of fair

This section provides a set of properties of the command **fair** culminating with Theorem 1 (fair-termination), which allows termination arguments to be decoupled from fairness. The command **chaos** allows any non-aborting behaviour. If a command refines **chaos**, that command is therefore non-aborting. The command **fair** is non-aborting.

Lemma 9 (chaos-fair). chaos $\sqsubseteq$ fair

Proof. The proof uses the definition of **chaos** (5), (54), the property that $c^\omega \sqsubseteq c^\star$, for any command c, and the definition of **fair** (3).

$$\mathbf{chaos} = (\epsilon \sqcap \pi)^\omega = \epsilon^\omega \ (\pi\ \epsilon^\omega)^\omega \sqsubseteq \epsilon^\star \ (\pi\ \epsilon^\star)^\omega = \mathbf{fair} \quad \Box$$

Fair execution of a command is always a refinement of the command.

Lemma 10 (introduce-fair). $c \sqsubseteq c \Cap \mathbf{fair}$

Proof. The lemma holds because **chaos** is the identity of $\Cap$ and Lemma 9 (chaos-fair):

$$c = c \Cap \mathbf{chaos} \sqsubseteq c \Cap \mathbf{fair}\ . \quad \Box$$

Fair execution followed by fair execution is equivalent to fair execution.

Lemma 11 (fair-fair). fair fair = fair

Proof.

$$\begin{array}{ll}
 & \mathbf{fair}\ \mathbf{fair} \\
= & \text{by definition of } \mathbf{fair} \text{ (3)} \\
 & \epsilon^\star\ (\pi\ \epsilon^\star)^\omega\ \epsilon^\star\ (\pi\ \epsilon^\star)^\omega \\
= & \text{by (53)} \\
 & (\epsilon^\star\ \pi)^\omega\ \epsilon^\star\ \epsilon^\star\ (\pi\ \epsilon^\star)^\omega \\
= & \text{as } c^\star\ c^\star = c^\star\text{, for any } c \\
 & (\epsilon^\star\ \pi)^\omega\ \epsilon^\star\ (\pi\ \epsilon^\star)^\omega
\end{array}
\qquad
\begin{array}{ll}
= & \text{by (53)} \\
 & \epsilon^\star\ (\pi\ \epsilon^\star)^\omega\ (\pi\ \epsilon^\star)^\omega \\
= & \text{as } c^\omega\ c^\omega = c^\omega\text{, for any } c \\
 & \epsilon^\star\ (\pi\ \epsilon^\star)^\omega \\
= & \text{by definition of } \mathbf{fair} \text{ (3)} \\
 & \mathbf{fair} \quad \Box
\end{array}$$

Fair execution of a sequential composition is implemented by fair execution of each command in sequence.

Lemma 12 (fair-distrib-seq). $(c\ d) \Cap \mathbf{fair} \sqsubseteq (c \Cap \mathbf{fair})\ (d \Cap \mathbf{fair})$

Proof. The proof uses Lemma 11 (fair-fair) and then interchanges weak conjunction with sequential (35).

$$(c\ d) \Cap \mathbf{fair} = (c\ d) \Cap (\mathbf{fair}\ \mathbf{fair}) \sqsubseteq (c \Cap \mathbf{fair})\ (d \Cap \mathbf{fair}) \quad \Box$$

The command **skip** $(= \epsilon^\omega)$ is the identity of parallel composition. It allows any sequence of environment steps, including ϵ^∞, but fair execution of **skip** excludes ϵ^∞, leaving only a finite sequence of environment steps: $\epsilon^\star$.

Lemma 13 (skip-fair). $\mathbf{skip} \Cap \mathbf{fair} = \epsilon^\star$

Proof. Expanding the definitions of **skip** (11) and **fair** (3) in the left side to start.

$$\begin{array}{ll}
& \epsilon^\omega \Cap \epsilon^\star\ (\pi\ \epsilon^\star)^\omega \\
= & \text{by Lemma 2 (finite-omega-prefix)} \\
& \epsilon^\star\ ((\mathbf{nil} \Cap (\pi\ \epsilon^\star)^\omega) \sqcap (\mathbf{nil} \Cap \epsilon\ \epsilon^\star\ (\pi\ \epsilon^\star)^\omega) \sqcap (\epsilon\ \epsilon^\omega \Cap (\pi\ \epsilon^\star)^\omega)) \\
= & \text{by (31) and Lemma 4 (iterate-pi-sync-atomic) and (40)} \\
& \epsilon^\star\ (\mathbf{nil} \sqcap \top \sqcap \top) \\
= & \epsilon^\star \quad \square
\end{array}$$

The command $\mathbf{term}$ (8) allows only a finite number of program steps but does not exclude an infinite sequence of environment steps, whereas **fair** excludes an infinite sequence of environment steps. When $\mathbf{term}$ and **fair** are conjoined, only a finite number of steps is allowed overall.

Lemma 14 (term-fair). $\mathbf{term} \Cap \mathbf{fair} = \alpha^\star$

Proof. Note that $\alpha^\star = \alpha^\star\ \alpha^\star \sqsubseteq \alpha^\star\ \epsilon^\star \sqsubseteq \alpha^\star\ \mathbf{nil} = \alpha^\star$, and hence $\alpha^\star = \alpha^\star\ \epsilon^\star$.

$$\begin{array}{ll}
& \mathbf{term} \Cap \mathbf{fair} = \alpha^\star \\
\Leftrightarrow & \text{by the definition of } \mathbf{term} \text{ (8) and } \alpha^\star = \alpha^\star\ \epsilon^\star \\
& \alpha^\star\ \epsilon^\omega \Cap \mathbf{fair} = \alpha^\star\ \epsilon^\star
\end{array}$$

The fixed point fusion theorem [1] is applied with $F \mathrel{\widehat=} \lambda x \cdot x \Cap \mathbf{fair}$, $G \mathrel{\widehat=} \lambda x \cdot \epsilon^\omega \sqcap \alpha\ x$ and $H \mathrel{\widehat=} \lambda x \cdot \epsilon^\star \sqcap \alpha\ x$. The lemma corresponds to $F(\nu G) = \nu H$, which holds by the fusion theorem if $F \circ G = H \circ F$ and F distributes arbitrary nondeterministic choices.

$$\begin{array}{ll}
& (F \circ G)(x) \\
= & \text{by the definitions of } F \text{ and } G \\
& (\epsilon^\omega \sqcap \alpha\ x) \Cap \mathbf{fair} \\
= & \text{distributing} \\
& (\epsilon^\omega \Cap \mathbf{fair}) \sqcap (\alpha\ x \Cap \mathbf{fair}) \\
= & \text{by Lemma 13 (skip-fair) and expanding the definition of } \mathbf{fair} \text{ (3)} \\
& \epsilon^\star \sqcap (\alpha\ x \Cap \epsilon^\star\ (\pi\ \epsilon^\star)^\omega) \\
= & \text{by unfolding (46) on } \epsilon^\star \text{ and distribute} \\
& \epsilon^\star \sqcap (\alpha\ x \Cap (\pi\ \epsilon^\star)^\omega) \sqcap (\alpha\ x \Cap \epsilon\ \epsilon^\star\ (\pi\ \epsilon^\star)^\omega) \\
= & \text{by Lemma 4 (iterate-pi-sync-atomic) and } \alpha \Cap \pi = \pi \text{ and (33)} \\
& \epsilon^\star \sqcap \pi\ (x \Cap \epsilon^\star\ (\pi\ \epsilon^\star)^\omega) \sqcap \epsilon\ (x \Cap \epsilon^\star\ (\pi\ \epsilon^\star)^\omega) \\
= & \text{distribute and use definition of } \mathbf{fair} \text{ (3)} \\
& \epsilon^\star \sqcap \alpha\ (x \Cap \mathbf{fair}) \\
= & \text{by the definitions of } H \text{ and } F \\
& (H \circ F)(x)
\end{array}$$

Finally F distributes arbitrary nondeterministic choices because for nonempty C,

$$F(\textstyle\bigsqcap C) = (\bigsqcap C) \Cap \mathbf{fair} = \displaystyle\bigsqcap_{c \in C} (c \Cap \mathbf{fair}) = \bigsqcap_{c \in C} F(c)\ ,$$

and for C empty, $F(\bigsqcap \emptyset) = \top \Cap \mathbf{fair} = \top = \bigsqcap_{c \in \emptyset}(c \Cap \mathbf{fair}) = \bigsqcap_{c \in \emptyset} F(c)$ because $\mathbf{chaos} \sqsubseteq \mathbf{fair}$. $\square$

We do not build fairness into our definitions of standard sequential programming constructs such as assignment, conditionals and loops [3], rather their definitions allow preemption by their environment forever. Hence any executable sequential program code may be preempted forever. The command $\mathbf{term}$ allows only a finite number of program steps but also allows preemption by the environment forever. If a command c refines $\mathbf{term}$ it will terminate in a finite number of steps provided it is not preempted by its environment forever, and hence fair execution of c only allows a finite number of steps because preemption by the environment forever is precluded by fair execution. That allows one to show termination by showing the simpler property, $\mathbf{term} \sqsubseteq c$, which does not need to consider fairness. Existing methods for proving termination can then be used in the context of fair parallel.

Theorem 1 (fair-termination). *If* $\mathbf{term} \sqsubseteq c$*, then* $\alpha^\star \sqsubseteq c \Cap \mathbf{fair}$.

Proof. If $\mathbf{term} \sqsubseteq c$, by Lemma 14 (term-fair) $\alpha^\star = \mathbf{term} \Cap \mathbf{fair} \sqsubseteq c \Cap \mathbf{fair}$. □

5 Properties of fair and concurrency

This section provides a set of properties for combining **fair** with (unfair) concurrency, in particular it provides lemmas for distributing fairness over a parallel composition. Details of abbreviated proofs can be found in [7]. The following is a helper lemma for Lemma 16 (fair-par-fair).

Lemma 15 (fair-par-fair-expand). $\mathbf{fair} \parallel \mathbf{fair} \; = \; \epsilon^\star \, (\mathbf{nil} \sqcap \pi \, (\mathbf{fair} \parallel \mathbf{fair}))$

Proof. The proof begins by expanding the definition of **fair** (3), then uses Lemma 1 (finite-finite-prefix) and (29), then Lemma 3 (iterate-pi-par-pi), Lemma 4 (iterate-pi-sync-atomic) and (29) and finally the definition of **fair** once more. □

Fair execution is implemented by fair execution of two parallel processes.

Lemma 16 (fair-par-fair). $\mathbf{fair} \sqsubseteq \mathbf{fair} \parallel \mathbf{fair}$

Proof.

$$\begin{array}{ll}
& \mathbf{fair} \sqsubseteq \mathbf{fair} \parallel \mathbf{fair} \\
\Leftrightarrow & \text{by the definition of } \mathbf{fair} \text{ (3) and (53)} \\
& (\epsilon^\star \, \pi)^\omega \, \epsilon^\star \sqsubseteq \mathbf{fair} \parallel \mathbf{fair} \\
\Leftarrow & \text{by (50)} \\
& \epsilon^\star \sqcap \epsilon^\star \, \pi \, (\mathbf{fair} \parallel \mathbf{fair}) \sqsubseteq \mathbf{fair} \parallel \mathbf{fair}
\end{array}$$

The above follows by Lemma 15 (fair-par-fair-expand) by distributing. □

Fair execution of $c \parallel d$ can be implemented by fair execution of each of c and d but the reverse does not hold in general.

Lemma 17 (fair-distrib-par-both). $(c \parallel d) \Cap \mathbf{fair} \sqsubseteq (c \Cap \mathbf{fair}) \parallel (d \Cap \mathbf{fair})$

Proof. The proof uses Lemma 16 (fair-par-fair) and then interchanges weak conjunction and parallel (45).

$$(c \parallel d) \Cap \mathbf{fair} \sqsubseteq (c \parallel d) \Cap (\mathbf{fair} \parallel \mathbf{fair}) \sqsubseteq (c \Cap \mathbf{fair}) \parallel (d \Cap \mathbf{fair}) \quad \Box$$

The following is a helper lemma for Lemma 19 (fair-par-chaos).

Lemma 18 (fair-par-chaos-expand). $\mathbf{fair} \parallel \mathbf{chaos} = \epsilon^\star\ (\mathbf{nil} \sqcap \pi\ (\mathbf{fair} \parallel \mathbf{chaos}))$

Proof. The proof uses the definitions of **fair** (3) and **chaos** (5) and (54), then Lemma 2 (finite-omega-prefix) and (29), then Lemma 3 (iterate-pi-par-pi) and Lemma 4 (iterate-pi-sync-atomic) and (29), and finally (54) and definitions (3) and (5). $\Box$

Fair execution in parallel with **chaos** gives a fair execution because **chaos** never aborts.

Lemma 19 (fair-par-chaos). $\mathbf{fair} \parallel \mathbf{chaos} = \mathbf{fair}$

Proof. The refinement from left to right is straightforward as $\mathbf{chaos} \sqsubseteq \mathbf{skip}$ and **skip** is the identity of parallel: $\mathbf{fair} \parallel \mathbf{chaos} \sqsubseteq \mathbf{fair} \parallel \mathbf{skip} = \mathbf{fair}$. The refinement from right to left uses the definition of **fair**.

$$\begin{array}{ll}
& \mathbf{fair} \sqsubseteq \mathbf{fair} \parallel \mathbf{chaos} \\
\Leftrightarrow & \text{by the definition of } \mathbf{fair} \text{ (3) and (53)} \\
& (\epsilon^\star\ \pi)^\omega\ \epsilon^\star \sqsubseteq \mathbf{fair} \parallel \mathbf{chaos} \\
\Leftarrow & \text{by (50)} \\
& \epsilon^\star \sqcap \epsilon^\star\ \pi\ (\mathbf{fair} \parallel \mathbf{chaos}) \sqsubseteq \mathbf{fair} \parallel \mathbf{chaos}
\end{array}$$

The above follows by Lemma 18 (fair-par-chaos-expand) and distributing. $\Box$

Fair execution of one process of a parallel composition eliminates behaviour ϵ^∞ for that process and hence because parallel compositions synchronise on ϵ (29), that eliminates behaviour ϵ^∞ from the parallel composition as a whole, provided the parallel process does not abort. Aborting behaviour of one process of a parallel aborts the whole parallel (36) and aborting behaviour allows any behaviour, including ϵ^∞. Fair execution of $c \parallel d$ can be implemented by fair execution of c (or by symmetry d).

Lemma 20 (fair-distrib-par-one). $(c \parallel d) \Cap \mathbf{fair} \sqsubseteq (c \Cap \mathbf{fair}) \parallel d$

Proof. The proof uses Lemma 19 (fair-par-chaos), then interchanges weak conjunction and parallel (45) and finally uses the fact that **chaos** is the identity of weak conjunction.

$$(c \parallel d) \Cap (\mathbf{fair} \parallel \mathbf{chaos}) \sqsubseteq (c \Cap \mathbf{fair}) \parallel (d \Cap \mathbf{chaos}) = (c \Cap \mathbf{fair}) \parallel d \quad \Box$$

6 Properties of fair parallel

This section examines the properties of the fair-parallel operator (16), such as commutativity, distribution over nondeterministic choice and associativity. The first three results derive readily from the equivalent properties for parallel.

Theorem 2 (fair-parallel-commutes). $c \parallel_f d = d \parallel_f c$

Proof. The proof is straightforward from definition (16) of fair-parallel because (unfair) parallel is commutative. □

Theorem 3 (fair-parallel-distrib). $D \neq \emptyset \Rightarrow c \parallel_f (\bigsqcap D) = \bigsqcap_{d \in D}(c \parallel_f d)$

Proof. Let D be non-empty.

$$\begin{aligned}
& c \parallel_f (\textstyle\bigsqcap D) \\
= \quad & \text{by the definition of } \parallel_f \text{ (16)} \\
& (c \Cap \mathbf{fair})\,\mathbf{skip} \parallel ((\textstyle\bigsqcap D) \Cap \mathbf{fair})\,\mathbf{skip} \\
= \quad & \text{as non-empty choice distributes over } \Cap\text{, sequential composition and parallel} \\
& \textstyle\bigsqcap_{d \in D}(c \Cap \mathbf{fair})\,\mathbf{skip} \parallel (d \Cap \mathbf{fair})\,\mathbf{skip} \\
= \quad & \text{by the definition of } \parallel_f \text{ (16)} \\
& \textstyle\bigsqcap_{d \in D}(c \parallel_f d) \quad \square
\end{aligned}$$

Theorem 4 (fair-par-monotonic). *If* $d_1 \sqsubseteq d_2$*, then* $c \parallel_f d_1 \sqsubseteq c \parallel_f d_2$.

Proof. The refinement $d_1 \sqsubseteq d_2$ holds if and only if $d_1 \sqcap d_2 = d_1$ and hence, by Theorem 3 (fair-parallel-distrib),

$$\begin{aligned}
& c \parallel_f d_1 \sqsubseteq c \parallel_f d_2 \\
\Leftrightarrow\ & c \parallel_f d_1 \sqcap c \parallel_f d_2 = c \parallel_f d_1 \\
\Leftrightarrow\ & c \parallel_f (d_1 \sqcap d_2) = c \parallel_f d_1
\end{aligned}$$

because $d_1 \sqcap d_2 = d_1$ follows from the assumption. □

Fair-parallel retains fairness for its component processes with respect to the overall environment even when one component process terminates.

Theorem 5 (fair-parallel-nil). $c \parallel_f \mathbf{nil} = (c \Cap \mathbf{fair})\,\mathbf{skip}$

Proof. The proof uses the definition of fair parallel (16), the facts that $\mathbf{nil} \Cap \mathbf{fair} = \mathbf{nil}$ and $\mathbf{skip}$ is the identity of parallel composition.

$$(c \Cap \mathbf{fair})\,\mathbf{skip} \parallel (\mathbf{nil} \Cap \mathbf{fair})\,\mathbf{skip} = (c \Cap \mathbf{fair})\,\mathbf{skip} \parallel \mathbf{skip} = (c \Cap \mathbf{fair})\,\mathbf{skip} \quad \square$$

While properties such as commutativity and distributivity are relatively straightforward to verify, associativity of fair-parallel is more involved. A property that is essential to the associativity proof is that fair-parallel execution of two commands not only ensures that each of its commands are executed fairly until program termination, but also that the whole parallel composition is executed fairly until program termination; this is encapsulated in Theorem 6 (absorb-fair-skip), but first we show the easy direction of this proof in Lemma 21 (introduce-fair-skip) and then give lemmas for the finite and infinite cases for Theorem 6 (absorb-fair-skip).

Lemma 21 (introduce-fair-skip). $c \parallel_f d \sqsubseteq ((c \parallel_f d) \Cap \mathbf{fair})\,\mathbf{skip}$

Proof.

$$\begin{array}{ll}
& c \parallel_f d \\
= & \text{by Lemma 8 (par-skip) using the definition of fair parallel (16)} \\
& (c \parallel_f d)\,\mathbf{skip} \\
\sqsubseteq & \text{by Lemma 10 (introduce-fair)} \\
& ((c \parallel_f d) \Cap \mathbf{fair})\,\mathbf{skip} \quad \square
\end{array}$$

Lemma 22 (finite-absorb-fair-skip).

$$(((c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\star)) \Cap \mathbf{fair})\,\mathbf{skip} = (c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\star)$$

Proof. The refinement from right to left follows by Lemma 21 (introduce-fair-skip). The refinement from left to right follows.

$$\begin{array}{ll}
& (((c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\star)) \Cap \mathbf{fair})\,\mathbf{skip} \\
= & \text{by the definition of } \parallel_f \text{ (16)} \\
& (((c \Cap \alpha^\star \Cap \mathbf{fair})\,\mathbf{skip} \parallel (d \Cap \alpha^\star \Cap \mathbf{fair})\,\mathbf{skip}) \Cap \mathbf{fair})\,\mathbf{skip} \\
\sqsubseteq & \text{by Lemma 17 (fair-distrib-par-both)} \\
& ((((c \Cap \alpha^\star \Cap \mathbf{fair})\,\mathbf{skip}) \Cap \mathbf{fair}) \parallel (((d \Cap \alpha^\star \Cap \mathbf{fair})\,\mathbf{skip}) \Cap \mathbf{fair}))\,\mathbf{skip} \\
\sqsubseteq & \text{by Lemma 12 (fair-distrib-seq) and } \Cap \text{ idempotent (38)} \\
& ((c \Cap \alpha^\star \Cap \mathbf{fair})\,(\mathbf{skip} \Cap \mathbf{fair}) \parallel (d \Cap \alpha^\star \Cap \mathbf{fair})\,(\mathbf{skip} \Cap \mathbf{fair}))\,\mathbf{skip} \\
\sqsubseteq & \text{as } \mathbf{skip} \Cap \mathbf{fair} = \epsilon^\star \text{ by Lemma 13 (skip-fair)} \\
& ((c \Cap \alpha^\star \Cap \mathbf{fair})\,\epsilon^\star \parallel (d \Cap \alpha^\star \Cap \mathbf{fair})\,\epsilon^\star)\,\mathbf{skip} \\
\sqsubseteq & \text{by Lemma 7 (sync-termination) as } \mathbf{skip} \parallel \mathbf{skip} = \mathbf{skip} \text{ and } \mathbf{skip} = \epsilon^\omega \\
& (c \Cap \alpha^\star \Cap \mathbf{fair})\,\mathbf{skip} \parallel (d \Cap \alpha^\star \Cap \mathbf{fair})\,\mathbf{skip} \\
= & \text{by the definition of } \parallel_f \text{ (16)} \\
& (c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\star) \quad \square
\end{array}$$

Lemma 23 (infinite-absorb-fair-skip).

$$(((c \Cap \alpha^\infty) \parallel_f d) \Cap \mathbf{fair})\,\mathbf{skip} = (c \Cap \alpha^\infty) \parallel_f d$$

Proof. The refinement from right to left follows by Lemma 21 (introduce-fair-skip). The refinement from left to right follows.

$$\begin{array}{ll}
& (((c \Cap \alpha^\infty) \parallel_f d) \Cap \mathbf{fair})\,\mathbf{skip} \\
= & \text{by the definition of } \parallel_f \text{ (16)} \\
& (((c \Cap \alpha^\infty \Cap \mathbf{fair})\,\mathbf{skip} \parallel (d \Cap \mathbf{fair})\,\mathbf{skip}) \Cap \mathbf{fair})\,\mathbf{skip} \\
= & \text{by Lemma 6 (infinite-annihilates)} \\
& (((c \Cap \alpha^\infty \Cap \mathbf{fair}) \parallel (d \Cap \mathbf{fair})\,\mathbf{skip}) \Cap \mathbf{fair})\,\mathbf{skip} \\
\sqsubseteq & \text{by Lemma 20 (fair-distrib-par-one) and } \Cap \text{ is idempotent (38)} \\
& ((c \Cap \alpha^\infty \Cap \mathbf{fair}) \parallel (d \Cap \mathbf{fair})\,\mathbf{skip})\,\mathbf{skip} \\
= & \text{by Lemma 6 (infinite-annihilates)} \\
& ((c \Cap \alpha^\infty \Cap \mathbf{fair})\,\mathbf{skip} \parallel (d \Cap \mathbf{fair})\,\mathbf{skip})\,\mathbf{skip} \\
= & \text{by Lemma 8 (par-skip)} \\
& (c \Cap \alpha^\infty \Cap \mathbf{fair})\,\mathbf{skip} \parallel (d \Cap \mathbf{fair})\,\mathbf{skip} \\
= & \text{by the definition of } \parallel_f \text{ (16)} \\
& (c \Cap \alpha^\infty) \parallel_f d \quad \square
\end{array}$$

Theorem 6 (absorb-fair-skip). $((c \parallel_f d) \Cap \mathbf{fair})\ \mathbf{skip} = c \parallel_f d$

Proof. The proof decomposes c and d into their finite and infinite components based on the observation that the identity of "$\Cap$" is **chaos**, which equals $\alpha^\star \sqcap \alpha^\infty$.

$$
\begin{array}{ll}
& ((c \parallel_f d) \Cap \mathbf{fair})\ \mathbf{skip} \\
= & \text{combine } \alpha^\star \sqcap \alpha^\infty \text{ with each of } c \text{ and } d \text{ and distribute} \\
& ((((c \Cap \alpha^\star) \sqcap (c \Cap \alpha^\infty)) \parallel_f ((d \Cap \alpha^\star) \sqcap (d \Cap \alpha^\infty))) \Cap \mathbf{fair})\ \mathbf{skip} \\
= & \text{by repeated application of Theorem 3 (fair-parallel-distrib)} \\
& (((c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\star)) \Cap \mathbf{fair})\ \mathbf{skip} \sqcap (((c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\infty)) \Cap \mathbf{fair})\ \mathbf{skip} \sqcap \\
& (((c \Cap \alpha^\infty) \parallel_f (d \Cap \alpha^\star)) \Cap \mathbf{fair})\ \mathbf{skip} \sqcap (((c \Cap \alpha^\infty) \parallel_f (d \Cap \alpha^\infty)) \Cap \mathbf{fair})\ \mathbf{skip} \\
= & \text{by Lemma 22 (finite-absorb-fair-skip) and Lemma 23 (infinite-absorb-fair-skip)} \\
& (c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\star) \sqcap (c \Cap \alpha^\star) \parallel_f (d \Cap \alpha^\infty) \sqcap \\
& (c \Cap \alpha^\infty) \parallel_f (d \Cap \alpha^\star) \sqcap (c \Cap \alpha^\infty) \parallel_f (d \Cap \alpha^\infty) \\
= & \text{by Theorem 3 (fair-parallel-distrib)} \\
& ((c \Cap \alpha^\star) \sqcap (c \Cap \alpha^\infty)) \parallel_f ((d \Cap \alpha^\star) \sqcap (d \Cap \alpha^\infty)) \\
= & \text{distributing} \\
& (c \Cap (\alpha^\star \sqcap \alpha^\infty)) \parallel_f (d \Cap (\alpha^\star \sqcap \alpha^\infty)) \\
= & \text{as } \alpha^\star \sqcap \alpha^\infty = \mathbf{chaos}\text{, the identity of } \Cap \\
& c \parallel_f d \quad \square
\end{array}
$$

With these results we can now verify associativity of fair parallel.

Theorem 7 (fair-parallel-associative). $(c \parallel_f d) \parallel_f e = c \parallel_f (d \parallel_f e)$

Proof.

$$
\begin{array}{ll}
& (c \parallel_f d) \parallel_f e \\
= & \text{by definition of } \parallel_f \text{ (16)} \\
& ((c \parallel_f d) \Cap \mathbf{fair})\ \mathbf{skip} \parallel (e \Cap \mathbf{fair})\ \mathbf{skip} \\
= & \text{by Theorem 6 (absorb-fair-skip)} \\
& (c \parallel_f d) \parallel (e \Cap \mathbf{fair})\ \mathbf{skip} \\
= & \text{by definition of } \parallel_f \text{ (16)} \\
& ((c \Cap \mathbf{fair})\ \mathbf{skip} \parallel (d \Cap \mathbf{fair})\ \mathbf{skip}) \parallel (e \Cap \mathbf{fair})\ \mathbf{skip} \\
= & \text{by associativity of parallel} \\
& (c \Cap \mathbf{fair})\ \mathbf{skip} \parallel ((d \Cap \mathbf{fair})\ \mathbf{skip} \parallel (e \Cap \mathbf{fair})\ \mathbf{skip}) \\
= & \text{by definition of } \parallel_f \text{ (16)} \\
& (c \Cap \mathbf{fair})\ \mathbf{skip} \parallel (d \parallel_f e) \\
= & \text{by Theorem 6 (absorb-fair-skip)} \\
& (c \Cap \mathbf{fair})\ \mathbf{skip} \parallel ((d \parallel_f e) \Cap \mathbf{fair})\ \mathbf{skip} \\
= & \text{by definition of } \parallel_f \text{ (16)} \\
& c \parallel_f (d \parallel_f e) \quad \square
\end{array}
$$

Other properties of fair parallel can be proven in a similar manner, for example, the equivalent of the interchange law (45) with parallel replaced by fair parallel.

7 Conclusions

Earlier work on fairness [16,13] focused on defining fairness as part of a fair-parallel operator. The main contribution of this paper is to separate the concerns of fairness and the parallel operator. That allows us to (i) reason about the fair execution of a single process in isolation, for example, via Theorem 1 (fair-termination); (ii) start from a basis of the (unfair) parallel operator, which has simpler algebraic properties; and (iii) define the fair-parallel operator in terms of the more basic (unfair) parallel operator and hence prove properties of the fair-parallel operator in terms of its definition.

The first point is important for devising a compositional approach to reasoning about the fairness properties of concurrent systems in terms of the fairness properties of their components. The second point allows us to utilise the synchronous concurrent refinement algebra [3,8,9] (which has similarities to Milner's SCCS [15,14]) to encode fairness in an existing theory with no built-in fair-parallel operator. The third point shows that no expressive power is lost compared to starting with a fair-parallel operator, in fact, there is a gain in expressiveness as one can define a parallel composition which imposes fairness on only one of its components: $((c \Cap \mathbf{fair})\ \mathbf{skip}) \parallel d$.

Overall, these results indicate that a suitable foundation of handling concurrency and fairness can start from a theory in which the parallel operator has no built-in fairness assumptions. The ability to do this derives from the use of a synchronous parallel operator motivated by the rely/guarantee approach of Jones [10,11,12] and Aczel's trace model for that approach [2,3,4,5], in which environment steps are made explicit.

Acknowledgements. This research was supported Australian Research Council Discovery Grant DP130102901. Thanks are due to Robert Colvin, Rob Van Glabbeek, Peter Höfner, Cliff Jones, and Kirsten Winter, for feedback on ideas presented here. This research has benefited greatly from feedback members of IFIP Working Group 2.3 on Programming Methodology, in particular, at the meeting in Villebrumier.

References

1. Chritiene Aarts, Roland Backhouse, Eerke Boiten, Henk Doombos, Netty van Gasteren, Rik van Geldrop, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. Fixed-point calculus. *Information Processing Letters*, 53:131–136, 1995. Mathematics of Program Construction Group.
2. P. H. G. Aczel. On an inference rule for parallel composition, 1983. Private communication to Cliff Jones `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/MSs/PHGA-traces.pdf`.
3. R. J. Colvin, I. J. Hayes, and L. A. Meinicke. Designing a semantic model for a wide-spectrum language with concurrency. *Formal Aspects of Computing*, 29:853–875, 2016.
4. F.S. de Boer, U. Hannemann, and W.-P. de Roever. Formal justification of the rely-guarantee paradigm for shared-variable concurrency: a semantic approach. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1245–1265. Springer Berlin / Heidelberg, 1999.
5. W.-P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

6. I. J. Hayes. Generalised rely-guarantee concurrency: An algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078, November 2016.
7. I. J. Hayes and L. A. Meinicke. Encoding fairness in a synchronous concurrent program algebra: extended version with proofs. arXiv:1805.01681 [cs.LO], 2018.
8. I.J. Hayes, R.J. Colvin, L.A. Meinicke, K. Winter, and A. Velykis. An algebra of synchronous atomic steps. In J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, editors, *FM 2016: Formal Methods: 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 352–369, Cham, November 2016. Springer International Publishing.
9. I.J. Hayes, L.A. Meinicke, K. Winter, and R.J. Colvin. A synchronous program algebra: a basis for reasoning about shared-memory and event-based concurrency. Ext. report at arXiv:1710.03352, 2017.
10. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Available as: Oxford University Computing Laboratory (now Computer Science) Technical Monograph PRG-25.
11. C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.
12. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM ToPLaS*, 5(4):596–619, 1983.
13. D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming: Eighth Colloquium Acre (Akko), Israel July 13–17, 1981*, pages 264–277, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.
14. A.J.R.G. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
15. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.
16. David Park. On the semantics of fair parallelism. In Dines Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer Berlin Heidelberg, 1980.
17. Rob J. van Glabbeek. Ensuring livenes properties of distributed systems (a research agenda). Technical report, NICTA, March 2016. Position paper.