

Fairness for Chorded Languages

Alexis Petrounias and Susan Eisenbach

Department of Computing
Imperial College London
alexisp@doc.ic.ac.uk and sue@doc.ic.ac.uk

Abstract. Joins or chords is a concurrency construct that seems to fit well with the object oriented paradigm. Chorded languages are presented with implicit assumptions regarding the fair treatment of processes by the scheduler. We define weak and strong fairness for the Small Chorded Object-Oriented Language (^lSCHOOL) which allows the classification of executions as fair. We investigate the liveness behaviour of programs and establish worst-case behaviours in terms of scheduling delays. We discover that weak fairness, although giving the scheduler implementer greater freedom in selecting the next process which is to be executed, is harder to implement than strong fairness; strong fairness benefits from a straightforward implementation, however, imposes many more constraints and limits the selection function of a scheduler.

1 Introduction

The chord construct is a concurrency mechanism inspired by the join from the Join-Calculus [7, 8]. Its use should raise the level of abstraction concurrent programs are written in, making development of correct programs easier. Its simplicity is appealing and recently languages have been extended to include chords or joins: C^\sharp [3], Concurrent Basic [12] an extension of Visual Basic, C_ω [11, 1], JoCaml [6], and Scala [5].

Generally, programmers assume that a concurrent execution environment will benefit from a “fair” scheduler, in the sense that execution of their programs will not be arbitrarily delayed, nor that some components of their program will be treated more favourably than others. However, this assumption is not typically reflected in the formalisms which describe concurrent languages and the selection function of the scheduler is left unspecified.

Notions of *fairness* evolved from the observation that legal executions allowed lack of progress for some components of concurrent systems. We aim to determine what makes a fair chorded implementation. To achieve this we start by defining a small calculus (^lSCHOOL) that is object oriented with chords.

Two fundamental concepts in scheduling are *liveness* and *fairness*, here in the sense of relative allocation (fairness) of execution time and message consumption among interacting processes which are capable of executing (liveness). There are many notions of fairness depending on the kinds of guarantees one wishes to give. We investigate two primary notions of fairness, *weak* and *strong*, and describe abstract schedulers with weak and strong fairness guarantees.

The rest of the paper is organised as follows: In section 2 we introduce both chords and fairness for earlier concurrent programming languages, showing some of the problems that arise from non-deterministic choice in programming language constructs, and the kinds of properties schedulers handling such choice may be required to guarantee. In section 3 we introduce our calculus. Fairness is investigated in sections 4,5, and 6. Finally, we conclude in section 7.

2 Background

Chorded programs [2] consist of classes which define chords. A chord consists of a header and a body. The header consists of at most one *synchronous* method signature and zero or more *asynchronous* method signatures, while the body consists of the expressions to be executed.

The body of a chord executes when an object has received an invocation for each of the method (signatures) in its header. In general, multiple invocations are required to execute the body. The simultaneous presence of invocations for each of the methods enables the chord to *join* [7, 8]. Hence a chord header can be seen as a guard for the execution of the body.

When a join occurs the participating asynchronous methods' invocations are consumed, and their arguments are passed to the body of the chord. When multiple invocations of the same method are present there is a non-deterministic choice as to which invocation is consumed.

A method can appear at most once in any chord header, but, methods can participate in multiple chord headers. If multiple chords can join by consuming the same method invocation, then the choice of which chord joins is unspecified.

The invocation of a synchronous method results in the invoking thread *blocking* until a suitable join occurs. Again, there may be a choice of which chord will join and unblock the thread if the method participates in more than one chord. Once the join occurs the invoker thread is unblocked and executes the chord body, potentially resulting in a return value.

Asynchronous methods return immediately, and their return type must be *async*, a subtype of *void*. A chord without a synchronous method is called asynchronous, and when it is capable of joining we call it *strung*; such chords will not have an invoking blocking thread. When an invocation for each of their asynchronous methods is present, the chord can join and its body is executed in a new thread. The following chord implements an unbounded buffer:

```
Object get( ) & async put( Object o ) { return o; }
```

Invoking `get`, which is synchronous, will result in the invoker blocking until a value is returned. The body of the chord will execute only when the chord joins, which requires an invocation of `put` to be present. When a join occurs, the invocation of `put` is consumed. Multiple invocations of `put` are “queued”.

The primary work on fairness in concurrent programming languages was for CCS by Costa and Stirling [4] who developed weak and strong fairness for CCS. Their approach is to augment CCS with an appropriate labelling mechanism

which deals with the non-deterministic choice operator $+$, as well as restriction and communication. The CCS operational semantics are extended so that only weakly-fair execution sequences are admitted in contrast with an approach where all executions are generated and unfair ones are then eliminated.

Weak fairness prohibits executions where a process remains enabled throughout but does not get a chance to proceed, or *remains enabled almost always*; in other words, weak fairness requires that *if a component is enabled continuously from some point onwards, then it eventually proceeds*. This implies that if a process is continuously enabled, then it proceeds infinitely often.

The resulting system allows for a “local” characterisation of weakly-fair executions, in the sense that finite sequences are shown to be weakly fair, and a continuous concatenation of such locally weakly-fair sequences produces a globally weakly-fair execution.

Strong fairness relaxes the assumption of weak fairness for a component becoming continuously enabled from some point onwards to becoming enabled infinitely often, and hence requires that *if a component is infinitely often enabled it proceeds infinitely often*. Therefore, strong fairness prohibits exactly those executions which contain components which become enabled infinitely often but proceed only a finite number of times.

Similarly to their presentation of weak fairness, a positive approach is used by which CCS is appropriately labelled and the operational semantics are extended in order to admit exactly the strongly-fair executions. In contrast with weak fairness, however, strongly-fair execution sequences cannot be characterised “locally”, as there is a family of systems of processes which are strongly-fair for an indefinite number of steps, yet then become inadmissible under strong fairness.

Strong fairness implies weak fairness, and if components which become disabled never become enabled again, the two coincide [10]. An implementation of strong fairness always requires using queues [10].

3 ^lSCHOOL

^lSCHOOL is a small object-oriented language. The constructs of the language are limited to classes which define chords, and object instantiations of these classes which reside in a heap. Classes exist within a simple, single-inheritance hierarchy, and methods and chords can be overridden¹

The chord description of the previous section is based on Polyphonic C[#], which features the *async* return type, a subtype of *void*, to indicate asynchronous methods; this is not necessary for implementing asynchronous methods, as all methods of return type *void* can be executed asynchronously. ^lSCHOOL does not feature the *async* type; instead, a method which has a return type of *void* can be invoked either synchronously or asynchronously. Furthermore we require exactly one argument for each method *m*, which we call *m.x*. The value of the last expression evaluated in a body becomes the return value of the chord.

¹ The full set of definitions and lemmas for ^lSCHOOL can be found in Appendix A.

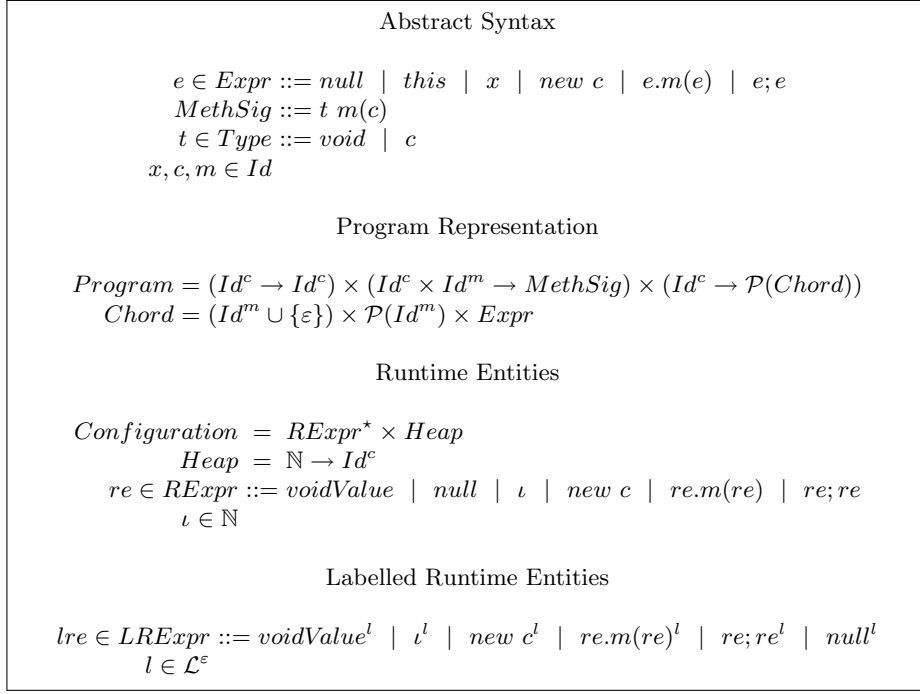


Fig. 1. ^lSCHOOL overview.

The granularity level of ^lSCHOOL is at the individual expressions, which are either live or non-live, depending on whether they can participate in an evaluation through one of the rules. Individual expressions are annotated with unique labels, and then transitions are annotated with collections of labels which correspond to those expressions participating in an evaluation. The use of labels thus enables us to observe execution traces, and classify these as admissible or inadmissible under particular notions of fairness.

Figure 1 provides an overview of syntax and program representation. The syntax of ^lSCHOOL expressions, *Expr*, consists of method calls $e.m(e)$, variables x , object creation $new\ c$, sequential composition, the special receiver *this*, and the *null* value. We use Id^m for the set of method names, Id^c for the set of class names, and Id^x for the set of variable names. ^lSCHOOL programs are represented using tuples of mappings. Programs consist of three mappings: inheritance, method signatures, and chords.

The first component maps a class name to its direct superclass: $Id^c \rightarrow Id^c$ and the superclass of a class c is $P \downarrow_1(c)$. The second component maps a class name c and a method name m to the method's signature: $Id^c \times Id^m \rightarrow MethSig$ and the method signature is $P \downarrow_2(c, m)$. The signature consists of a return type, a name, and the class type of the single argument: $t\ m(c)$. The third component maps a class name to the set of chords defined in the class: $Id^c \rightarrow$

$\mathcal{P}(\text{Chord})$; to obtain the chords of class c we use $P\downarrow_3(c)$. We encode chords as triplets $(Id^m \cup \{\varepsilon\}) \times \mathcal{P}(Id^m) \times Expr$. The first element is either the name of a synchronous method, or the symbol ε if there aren't any. The second element is a set of asynchronous method names. The third is the body of the chord.

Each expression is annotated with a distinct label which is a runtime entity. Expressions which are ground values, so not considered during future evaluation steps, are each labelled with a special *empty* label.

Only top-level run-time expressions, $RExpr$, are annotated, forming the labelled run-time expressions, $LRExpr$, and so all sub-expressions come directly from the ^lSCHOOL runtime expressions $RExpr$ and have no labels.

Execution of ^lSCHOOL expressions is described by a term rewriting system in which a configuration, consisting of a collection of expressions, e_i , and a heap, h , evaluate into a new configuration:

$$e_1, \dots, e_n, h \longrightarrow e'_1, \dots, e'_m, h'$$

where the heap is a mapping of object addresses to their class names:

$$h : \mathbb{N} \rightarrow Id^c$$

and the number of expressions may change from n to m , as new expressions are *spawned* when asynchronous chords join and their bodies execute. Furthermore, threads never terminate in the sense that ground expressions are not removed from the execution, so $m \geq n$.

We also use the shorthand \bar{e} for several, concurrent expressions, and thus we also have:

$$\bar{e}, h \longrightarrow \bar{e}', h'$$

We describe ^lSCHOOL using operational semantics found in figure 2; we use the variable v to designate acceptable values for arguments to method calls, which consist of all expressions other than *voidValue*, and the variable z to designate all irreducible values (*null*, ι , *voidValue*). The evaluation rules are:

- **NEW**: creates a new object of a given class and allocates a previously undefined heap address which now maps to the object; returning the new address.
- **SEQ**: discards an irreducible value and enables the evaluation of the next expression in a sequential composition; the final value in a sequential composition cannot be discarded, and this allows the final value of a chord's body to become the return value of the chord's synchronous method.
- **ASYNC**: places an asynchronous invocation of a method (of *void* type and appearing in at least one asynchronous part of a chord header) into the execution, available for joining later. The invocation immediately returns *voidValue*. The condition $E[\cdot] \neq [\cdot]$, requiring the evaluation context to not be empty, is necessary so that we avoid infinite reductions of the form:

$$\begin{aligned} \iota.m(v), h &\longrightarrow voidValue, \iota.m(v), h \longrightarrow \\ &voidValue, voidValue, \iota.m(v), h \longrightarrow \dots \end{aligned}$$

$\frac{h(\iota) \text{ is undefined} \quad \iota' \text{ is fresh}}{E[\text{new } c]^\iota, h \xrightarrow{\{\iota\}} E[\iota]^\iota, h[\iota \mapsto c]} \text{NEW}$	$\frac{\iota' \text{ is fresh}}{E[z; e]^\iota, h \longrightarrow E[e]^\iota, h} \text{SEQ}$
$\frac{h(\iota)=c \quad m \in \bigcup_{\chi \in P \downarrow_3(c)} \chi \downarrow_2 \quad E[\cdot] \neq [\cdot] \quad \iota', \iota'' \text{ are fresh}}{E[\iota.m(v)]^\iota, h \xrightarrow{\{\iota\}} E[\text{voidValue}]^{\iota'}, \iota.m(v)^{\iota''}, h} \text{ASYNC}$	
$\frac{h(\iota)=c \quad (m, \{m_1, \dots, m_k\}, e) \in P \downarrow_3(c) \quad \iota'_0 \text{ fresh}}{\forall i \in 1..k : \exists \iota'_i \in \mathcal{L}^\varepsilon : \iota'_i = \begin{cases} \varepsilon & \text{if } E_i[\cdot] = [\cdot] \\ \iota \in \mathcal{L}, \iota \text{ fresh} & \text{otherwise} \end{cases}} \text{JOIN}$	
$\frac{E[\iota.m(v)]^{\iota'_0}, E_1[\iota.m_1(v_1)]^{\iota'_1}, \dots, E_k[\iota.m_k(v_k)]^{\iota'_k}, h \xrightarrow{\{\iota'_0, \dots, \iota'_k\}} E[e^\iota/\text{this}, v/m_x, v_1/m_{1-x}, \dots, v_k/m_{k-x}]^{\iota'_0}, E_1[\text{voidValue}_1]^{\iota'_1}, \dots, E_k[\text{voidValue}_k]^{\iota'_k}, h}{\text{STRUNG}}$	
$\frac{E_1[\iota.m_1(v_1)]^{\iota'_1}, \dots, E_k[\iota.m_k(v_k)]^{\iota'_k}, h \xrightarrow{\{\iota'_1, \dots, \iota'_k\}} E_1[\text{voidValue}_1]^{\iota'_1}, \dots, E_k[\text{voidValue}_k]^{\iota'_k}, E[e^\iota/\text{this}, v_1/m_{1-x}, \dots, v_k/m_{k-x}]^\iota, h}{\text{PERM}}$	
$\frac{\bar{e} \cong \overline{e^{\iota''} e^{\iota''\iota''}} \quad \overline{e^{\iota''}}, h \xrightarrow{\mu} \overline{e^{\iota''\iota''}}, h' \quad \overline{e^{\iota''\iota''} e^{\iota''\iota''}} \cong \overline{e^{\iota''}}}{\bar{e}^\iota, h \xrightarrow{\mu} \bar{e}^{\iota'}, h'} \text{PERM}$	

Fig. 2. ^lSCHOOL operational semantics.

- **JOIN**: selects a chord in with a live synchronous method and *joins* this chord, consuming the corresponding asynchronous invocations (and replacing each by *voidValue*). The actual arguments are mapped to the formal arguments of the chord's body, which becomes the new evaluating expression.
- **STRUNG**: selects an asynchronous chord which is *strung*, i.e can join. Similar to the JOIN rule, all the asynchronous invocations are consumed, and actual arguments are mapped to the formal arguments of the chord's body, which will execute concurrently with the rest of the expressions.
- **PERM** enables the non-deterministic selection of expressions to evaluate and the reordering of expressions in the execution. The notation $\bar{e} \cong \bar{e}'$ means that \bar{e}' is a permutation of \bar{e} .

The selection of which *strung* chord to join happens at two levels: multiple receiver objects may have asynchronous invocations enabling the joining of a chord, and an object may feature multiple chords which currently can join.

```

1 class Example {
2   void f() & async a() { b(); }
3   void g() & async b() { print "Hi"; }
4 }

```

Listing 1.1. Example of class for semantics.

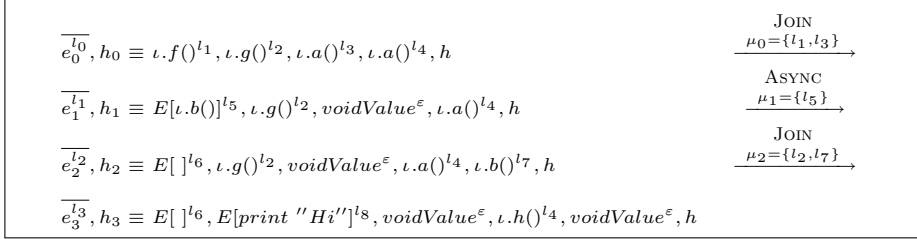


Fig. 3. An execution of the `Example` class in ^lSCHOOL.

At each step there is a set of labels for the expressions affected by the rule used, the *participating* labels. Ground values become annotated with the empty label, and each changed or new expression gets a *fresh* label. Labels which may participate in an evaluation step are termed *active*. The freshness of a label is a property of the entire execution sequence up to the first appearance of the label. The participating labels, μ , form a set. Furthermore, the participating labels cannot be empty because each rule changes at least one expression from the initial configuration. Also, the empty label cannot be a participating label. Finally, participating labels are consumed.

Consider an example of an execution in figure 3 using the class `Example` from listing 1.1 with two chords: the first requires an asynchronous invocation of `a` in order to join with a synchronous invocation of `f`, while the second requires an asynchronous invocation of `b` in order to join with a synchronous invocation of `g`. Initially there exists a sole object of class `Example` at address ι , and four invocations of methods `f`, `g` and two of `a` respectively.

The first step of evaluation joins the first chord using the JOIN rule on the labels l_1 and l_3 , which form the participating labels of this step, μ_0 . These labels were consumed. The consuming of the asynchronous invocation of `a` results in its replacement with *voidValue*, and its annotation with the empty label. The body of the chord forms a new expression, and is annotated with the new label l_5 . The label l_2 , which did not participate in the evaluation step, appears in the resulting configuration. The next two steps of evaluation result in the second chord joining, and thus the eventual participation of l_2 . The second invocation of `a`, labelled l_4 , never participated in the execution, and so the label remains. The labels of a configuration remain finite, and there is an upper bound on the creation of new expressions, and hence new labels. The *live* labels of a configuration, $e_1^{l_1}, \dots, e_n^{l_n}, h$, are those labels which can participate in the next

```

1 class LivenessExample {
2   void f() & async a() { b(); }
3   void g() & async b() { print "Hi"; }
4   void h() & async b() { print "I feel ignored"; }
5 }

```

Listing 1.2. Example class for liveness of labels.

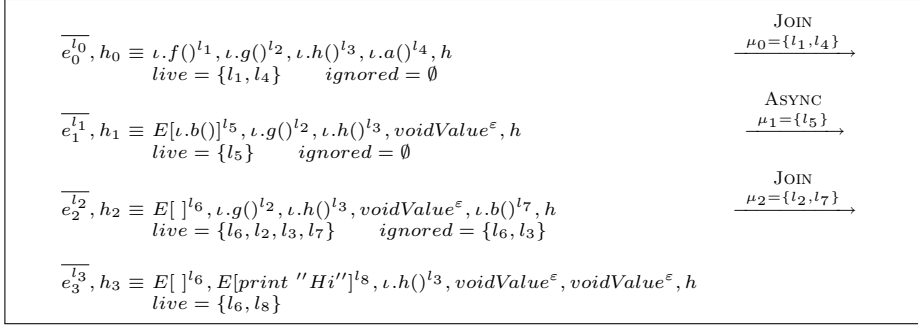


Fig. 4. An execution of the `LivenessExample` class in ^lSCHOOL.

evaluation step, $\xrightarrow{\mu}$. A label is live when there is at least one rule through which it can participate in the next evaluation step.

An evaluation rule may be applicable for several sets of participating labels, more than one of which may contain the live label under consideration. Also, more than one rule through which the label can participate may be applicable. The former case would hold if the label's underlying expression is a method invocation which can currently participate through the JOIN rule in two different chords. The latter case would hold if the expression can currently participate either through the JOIN rule or the STRUNG rule, again in two different chords.

At each step of an execution, if a label, l , is live, but is not in the participating labels, μ , then we say it is *ignored*. A label loses its liveness when it participates in a step of evaluation. However, a label can also lose its liveness due to another label being consumed (such as two labels competing for a sole third label in order to join). Conversely, a label may become live due to a newly created label (such as a method invocation to join).

Consider an example of liveness in figure 4 using the class `LivenessExample` from listing 1.2, which features three chords. The second and third chord both require an invocation of `b` in order to join, and hence will compete for such an invocation. Initially there exists a sole object of class `LivenessExample` at address ι , and four invocations: three synchronous invocations of `f`, `g` and `h` respectively, and an asynchronous invocation of `a`.

We notice that after two steps of evaluation both the second and third chords can join, as both their synchronous method invocations, `g` and `h`, respectively,

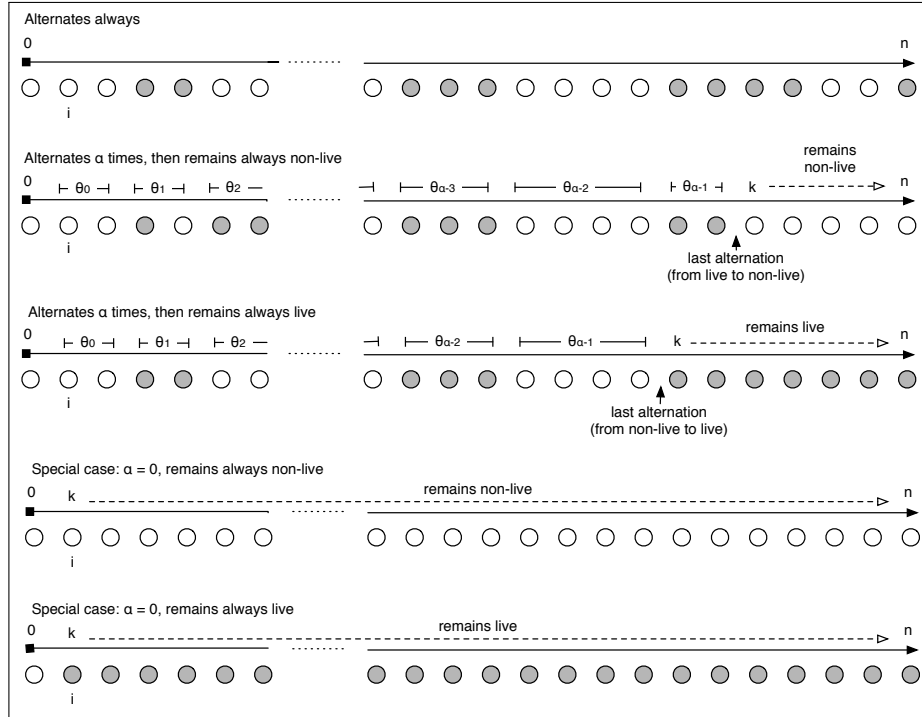


Fig. 5. Liveness behaviour of a label: alternation – label live at grey circles.

can participate through the JOIN rule and consume the invocation of **b**; however, during the third step the invocation of **g** was selected to participate, and the invocation of **h** was ignored, resulting in loss of liveness (as there are no more asynchronous invocations of **b**).

The losing and regaining of liveness of a label (its *liveness behaviour*) is subject to the selection of evaluation rule applied at each step of an execution. It is possible to classify liveness behaviour in terms of patterns exhibited, and accordingly make observations on the properties of these patterns.

Figure 5 illustrates the possible liveness behaviours of a label: always alternating, alternating α times and then remaining always non-live or always live, and the two special cases when $\alpha = 0$.

We are interested in establishing the worst case for the number of configurations - or steps - which occur after a given number of alternations. Hence, if we assume that each θ_m is equal to θ , then we know that in the worst case, and assuming that l starts out as non-live, the last alternation will occur before the $i + \alpha * \theta$ configuration. The bottom example execution of figure 6 illustrates this pattern for l starting out as non-live and after α alternations remaining always live; the same calculation of the worst case holds for other combinations of initial and final liveness of l .

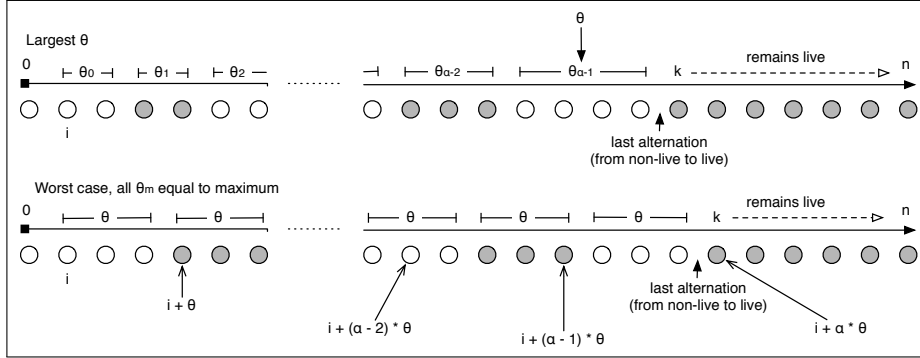


Fig. 6. Liveness behaviour of a label: worst case for the number of configurations which occur before a label alternates for the last time – label live at grey circles.

$l \in \text{live}(\overline{e_i^{l_i}})$	α	Worst Case Number of Steps
no	even	$i + (\alpha/2) * (\theta + 1)$
no	odd	$i + \lceil \alpha/2 \rceil * (\theta + 1) - 1$
yes	even	$i + (\alpha/2) * (\theta + 1)$
yes	odd	$i + \lfloor \alpha/2 \rfloor * (\theta + 1) + 1$

Table 1. Summary of worst-case calculations for number of steps which occur when ignoring a label is maximised.

Furthermore, we are interested in the worst case for the number of configurations - or steps - which occur when we maximise the number of times a label is ignored during finite alternating behaviour. In order for l to be ignored it must be live, and hence we consider the case where l is live for only one configuration before it alternates to non-live again, and each sequence of configurations at which l is non-live is equal to the maximum, θ . In order to consider the worst case, however, we have to take into account whether α is even or odd, and whether l starts out as live non-live. Figure 7 illustrates the four cases which arise, and table 1 summarises the calculations.

4 Definitions of Weak and Strong Fairness

Since a label is ignored only when it is live and does not participate in the current step of evaluation, and only active labels can become live, the following definitions of fairness are stated in terms of live labels, ignoring empty labels.

Definition 1 (Weak Fairness).

An execution is weakly-fair iff no process remains ignored continuously. In other words, no process remains live continuously. [4, 10, 9]

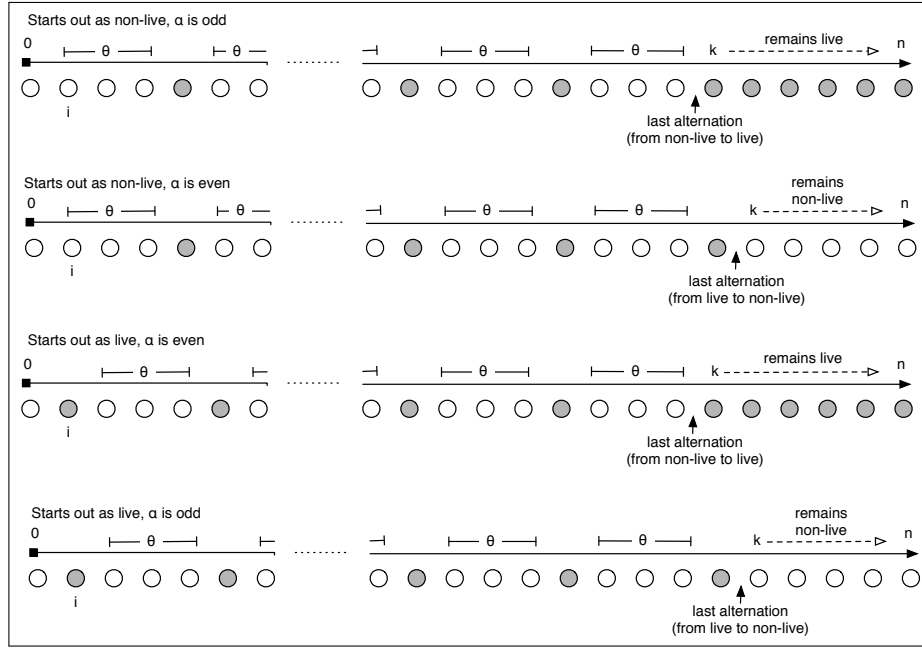


Fig. 7. Liveness behaviour of a label: maximising the number of times a label is ignored.

```

1 class WeakFairnessExample {
2   void f() & async a() { b(); a(); f(); }
3   void g() & async b() { print "Oh, dear..."; }
4   void h() & async a() { print "Help!"; }
5   void k() & async c() { print "Ha!"; }
6 }

```

Listing 1.3. Example class for weak fairness.

From this definition, every label must cease to be live after a finite number of steps. The following definition requires that for every label appearing in an execution, there exists some configuration at which the label is not live.

Definition 2 (Weakly-Fair Execution).

$$\begin{aligned}
\overline{e_0^l}, h_0 &\xrightarrow{\mu_0} \overline{e_1^l}, h_1 \xrightarrow{\mu_1} \overline{e_2^l}, h_2 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^l}, h_n \xrightarrow{\mu_n} \dots \implies \\
&\forall i \in \mathbb{N} : \forall l \in \text{labels}(\overline{e_i^l}) : \exists k \geq i : l \notin \text{live}(\overline{e_k^l})
\end{aligned}$$

Consider an example execution inadmissible under weak fairness using the class from listing 1.3 and the initial configuration from figure 8. The class features four chords: the first and third require the joining, respectively, of the synchronous methods **f** and **h** with the asynchronous method **a**, the second chord requires the joining of the synchronous method **g** with the asynchronous

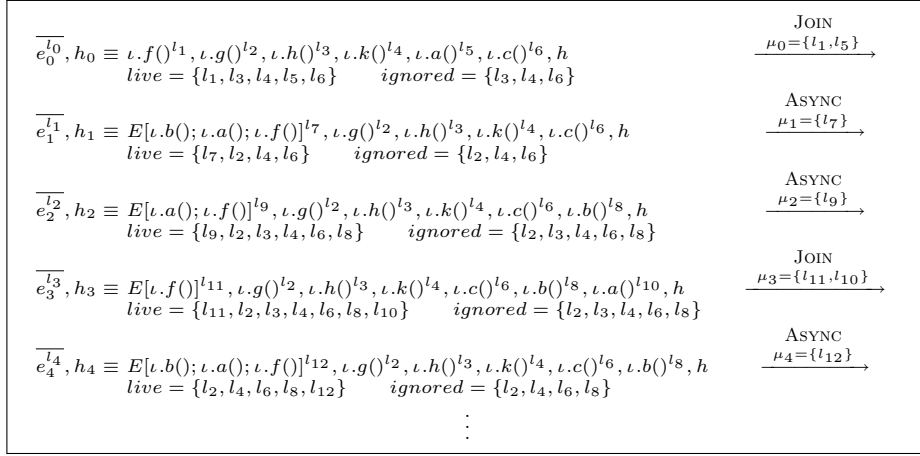


Fig. 8. An execution inadmissible under weak fairness.

method **b**, and finally the fourth chord requires the synchronous method **k** to join with the asynchronous method **c**. Initially, there exists a sole object of class `WeakFairnessExample` at address ι , and six invocations of the methods **f**, **g**, **h**, **k**, **a** and **c** respectively (there is no invocation of **b**).

In the initial configuration all invocations except that of **g**, labelled by l_2 , are live. The first invocation, labelled l_1 , is selected to join with l_5 , and hence the rest (l_3, l_4, l_6) are ignored. At the second configuration the sole invocation of **a** has been consumed, and hence the third chord cannot join any more, resulting in l_3 losing its liveness. Furthermore, the invocation of **g** becomes live now, as now it is possible to apply the JOIN rule with the invocation of **b** from within the evaluation context of l_7 ; instead, l_7 is selected to execute via the ASYNC rule, resulting in **b** entering the configuration in the next configuration with label l_8 .

At this point (configuration 2) l_3 becomes live again, as it is possible to apply the JOIN rule; once again, the ASYNC is chosen with l_9 and the invocation of **a** enters the configuration with label l_{10} . The fourth chord is ignored for a third consecutive time. At this point, it is possible for the execution to repeat itself in this pattern indefinitely, resulting in l_4 and l_6 being ignored continuously, and hence making the execution unfair under weak fairness. Label l_2 became live in the second step and remains live; so after the second step, execution is not weakly fair for l_2 . A weakly-fair execution sequence will print both “Ha!” and “Oh, dear...”, although “Help!” has no guarantee of ever printing.

Definition 3 (Strong Fairness).

An execution is strongly-fair iff no process is ignored infinitely-often. In other words, no process loses and regains its liveness an infinite number of times. [4, 10, 9]

From definition 3 every label must cease to be live after a finite number of steps, and never regain its liveness. The following definition requires that for

```

1 class StrongFairnessExample {
2   void f() & async a() { b(); f(); }
3   void f() & async b() { b(); f(); }
4   async b() { a(); }
5   void g() & async a() { print "Help"; }
6 }

```

Listing 1.4. Example class for strong fairness.

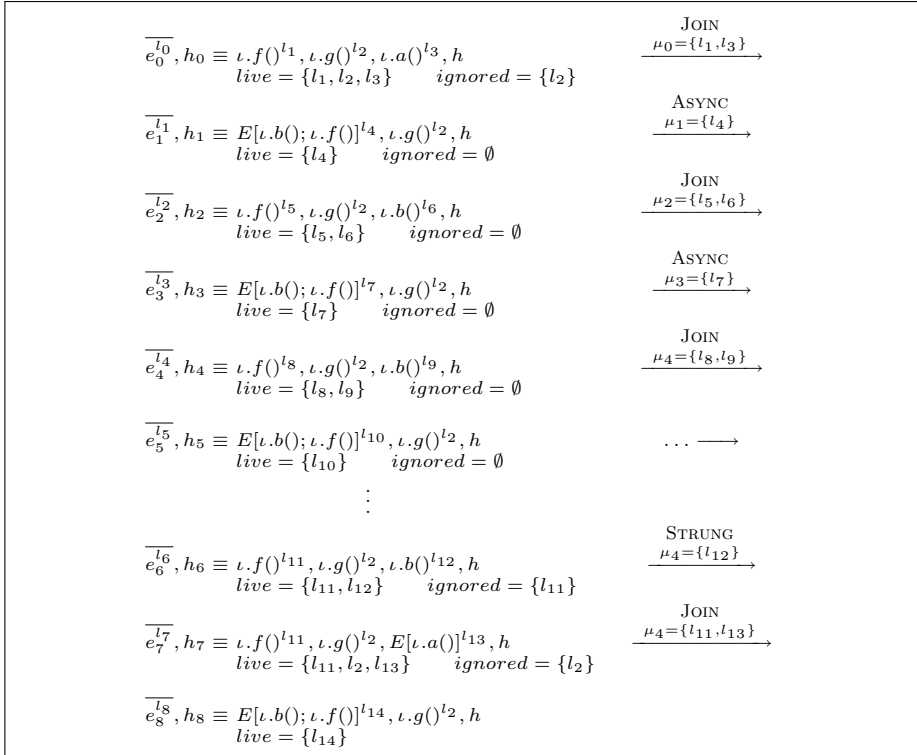


Fig. 9. An execution inadmissible under strong fairness.

every label appearing in an execution, there exists some configuration such that the label is not live at that and all further configurations.

Definition 4 (Strongly-Fair Execution).

$$\begin{aligned}
& \overline{e_0^{l_0}}, h_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1 \xrightarrow{\mu_1} \overline{e_2^{l_2}}, h_2 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow{\mu_n} \dots \implies \\
& \forall i \in \mathbb{N} : \forall l \in \text{labels}(\overline{e_i^{l_i}}) : \exists j \geq i : \forall k \geq j : l \notin \text{live}(\overline{e_k^{l_k}})
\end{aligned}$$

Consider an example execution inadmissible under strong fairness using the class from listing 1.4 and the initial configuration from figure 9. The class features

four chords: the first two require the joining of the synchronous method **f** with the asynchronous methods **a** and **b** respectively, the third chord is asynchronous and requires only the method **b**, and finally the fourth chord requires the synchronous method **g** to join with **a**. Initially, there exists a sole object of class `StrongFairnessExample` at address ι , and three invocations of the methods **f**, **g**, and **a** respectively.

In the initial configuration the invocation of **g**, labelled l_2 , is live. However, the first invocation, labelled l_1 , is selected to join with l_3 , and hence l_2 is ignored. In the second configuration l_2 has lost its liveness. The execution now continues for an unspecified number of steps during which the second chord joins repeatedly. Newly created labels which become live participate immediately, and since there is no availability of an invocation of **a**, l_2 does not regain its liveness. It is not possible, however, to claim that such an execution is strongly-fair because it is still possible for l_2 to regain its liveness. Label l_2 could lose and regain its liveness infinitely often, and hence be ignored infinitely often. Thus, a strongly-fair execution prohibits this example execution. If, however, l_2 were to participate in the step after the configuration indexed by 7 then the sequence would be admissible under strong fairness.

5 Weak Fairness

We describe a mechanism to realise weak-fairness constraints for ^lSCHOOL, showing how the example of our execution inadmissible under weak fairness needs to be constrained. Since no label remains live throughout a weakly-fair execution, starting from a configuration, each of its live labels must eventually lose its liveness. So we can consider a *localised* definition of weak fairness: a sequence is *locally weakly fair for the initially live labels* only if each of the labels is not live in at least one future configuration. Once the initial configuration's live labels have lost their liveness, we get a locally weakly-fair execution.²

The next locally weakly-fair execution uses the previous final configuration as the new initial configuration. The concatenation of two locally weakly-fair execution sequences is weakly fair. Also, for any locally weakly-fair execution sequence which does not result in termination, it is always possible to continue execution, and thus obtain a weakly-fair execution. Since executions can have an infinite length, a weakly-fair execution is then defined as *the maximal sequence of concatenated locally weakly-fair executions*.

One way to generate locally weakly-fair executions is to keep track of the *serviced* labels, those which have lost their liveness. Once all initially live labels are included in the set of serviced labels, we have obtained a locally weakly-fair execution sequence and begin anew. Therefore, our mechanism consists of a selection rule which allows one to freely select any applicable evaluation rule while keeping track of the serviced labels, and a concatenation mechanism which allows the construction of a weakly-fair execution sequence.

² Definitions, lemmas, and theorems for weak fairness can be found in Appendix B.

$$\begin{array}{c}
\overline{e^i}, h \xrightarrow{\mu} \overline{e^{i'}}, h' \\
L' \text{ largest subset of } \textit{live}(\overline{e^i}) : \\
L \cap L' = \emptyset \quad \wedge \quad L' \cap \textit{live}(\overline{e^{i'}}) = \emptyset \\
\hline
\overline{e^i}, h, L \xrightarrow{\mu} \overline{e^{i'}}, h', L \cup L' \quad \text{WEAK}
\end{array}$$

Fig. 10. Weakly-fair selection rule.

We define a weakly-fair execution as the maximal sequence of locally weakly-fair execution sequences; a locally weakly-fair execution sequence consists of weakly-fair evaluation steps, which are obtained through the WEAK selection rule of figure 10. The rule allows us to select any ^lSCHOOL evaluation rule which is applicable, while maintaining the set L of serviced labels. This set is built at each step by adding those labels which have lost their liveness; labels which have already been serviced are not placed into L again. The largest subset of live labels in the initial configuration is considered, resulting in a notion of completeness for the recording process. The WEAK is useful because it allows us to non-deterministically generate *all* weakly-fair schedules from a given initial configuration, some of which may have no finite upper bound on their length.

Each locally weakly-fair execution sequence, of some length k , begins with a finite configuration and an empty set of serviced labels L_0 and a finite set of initially live labels; at the final configuration all initially live labels are included in L_k . The sequence of participating labels is recorded in M . The size of L_i is a function of the number of labels which participate in the evaluation steps 0 through $i - 1$; although there is an upper bound λ on the number of new labels which can be created at each step, and hence the number of labels which be added to each L , the size of L_k has no upper bound.

We use the original example of weak fairness with the program of listing 1.3 and the initial configuration from figure 8. In figure 11 we generate a locally weakly-fair execution which is the same as the original execution up to the configuration indexed by 4; each L_i is recorded, with added labels underlined. L_0 begins empty, and L_1 contains labels l_1 , l_3 and l_5 , as the first and third have lost their liveness by participating in the first evaluation step, and l_3 has lost its liveness because the only label that could join with it, l_5 , was consumed. The execution then continues up to configuration 4 as in the original example.

At this point we see from the set of serviced labels L_4 that all original live labels except l_4 and l_6 have been serviced; furthermore, both these labels are live, and hence we may chose to apply the JOIN rule with the labels and obtain the last configuration, indexed by 5, in the figure. Now, all original live labels are contained in L_5 , and hence the execution sequence is locally weakly fair. Notice that we could have continued executing in the initial pattern an indefinite number of times before choosing to join l_4 and l_6 .

Starting from the last configuration (indexed by 5) of the locally weakly-fair execution sequence from above, we can begin a new locally weakly-fair execution

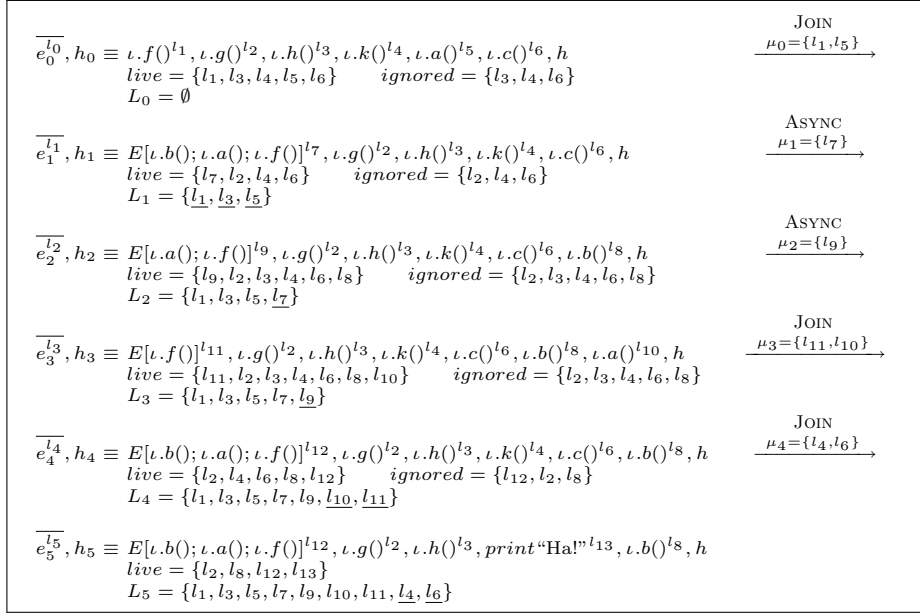


Fig. 11. A locally weakly-fair execution.

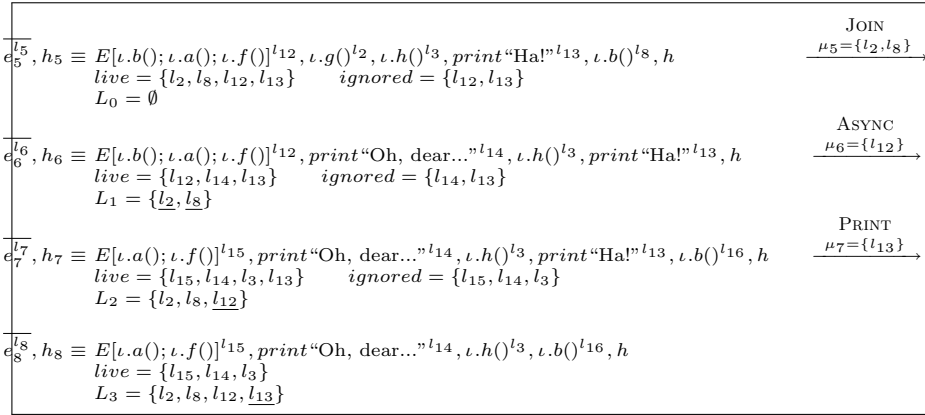


Fig. 12. Next locally weakly-fair execution to be concatenated.

sequence by aiming to service the live labels $\{l_2, l_8, l_{12}, l_{13}\}$; an example such sequence is in figure 12 where the second chord joins (labels l_2 and l_8), then the invocation of **b** from the context labelled by l_{12} is placed into the configuration through the **ASYNC** rule, and finally the **print** command, labelled by l_{13} , is executed via an unspecified but obvious **PRINT** rule. The last set of serviced labels, L_3 , is the set of original live labels, so this sequence is locally weakly-fair. The execution sequence indexed by 5 through 8 is also locally weakly-fair, and

$$\boxed{
\begin{array}{c}
\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h' \\
\forall l \in \text{live}(\overline{e^l}) : \exists l \in \text{live}(\overline{e^l}) : l \leq_Q l \wedge l \in \mu \\
Q' = Q \upharpoonright \text{active}(\overline{e^{l'}}) \circ \text{active}(\overline{e^{l'}}) \setminus \text{active}(\overline{e^l}) \\
\hline
\overline{e^l}, h, Q \xRightarrow{\mu} \overline{e^{l'}}, h', Q' \quad \text{STRONG}
\end{array}
}$$

Fig. 13. Strongly-fair selection rule.

so their concatenation is weakly-fair. The message “Ha!” was printed and the message “Oh, dear...” will eventually print if we continue execution with the next locally weakly-fair sequence, however, the message “Help!” may never print.

In order to show correctness for weakly-fair executions, we first establish correctness for locally weakly-fair executions, in the sense that a locally weakly-fair execution, for each initially live label, contains at least one configuration at which the label is not live. A weakly-fair execution sequence corresponds to a sequence of locally weakly-fair execution sequences, which in turn corresponds to an ^lSCHOOL execution sequence.

6 Strong Fairness

We describe a mechanism that realises ^lSCHOOL strong fairness constraints. The example from section 4 of an execution inadmissible under strong fairness is altered by imposing these constraints. We also present worst-case calculations for the liveness behaviour of labels under strong fairness.³

An execution sequence is strongly-fair when no label becomes live infinitely often. Since executions can be infinite, it is not possible to first generate all valid executions and then select those which are strongly-fair; so we employ a mechanism which *maintains* a strongly-fair execution as it is being generated.

For each application of an evaluation rule constraints are imposed on future selections, and accumulate throughout. To keep track of these constraints we introduce a *queue of labels*, which is modified at each evaluation step and passed to the next. Strong fairness is maintained by imposing constraints through the selection rule STRONG in figure 13: all new labels are appended to the queue, the order of labels remains unchanged, labels which participate in an evaluation are removed from the queue, and at each step the first live label in the queue always participates in the next evaluation step.

Starting from a finite initial configuration, a label which is repeatedly ignored will eventually reach the head of the queue. Once a label is at the head of the queue, it will be selected for participation the next time it becomes live. Since all labels are added to the queue, all labels eventually lose their liveness forever.

A queue Q is denoted by $\langle l_1 \circ \dots \circ l_k \rangle$, and its size k is denoted by $|Q|$. The relation $l \leq_Q l'$ between two labels, l and l' , appearing in a queue, Q , is defined either when $l = l'$, or when l appears before l' in Q .

³ The full set of definitions for strong fairness can be found in Appendix C.

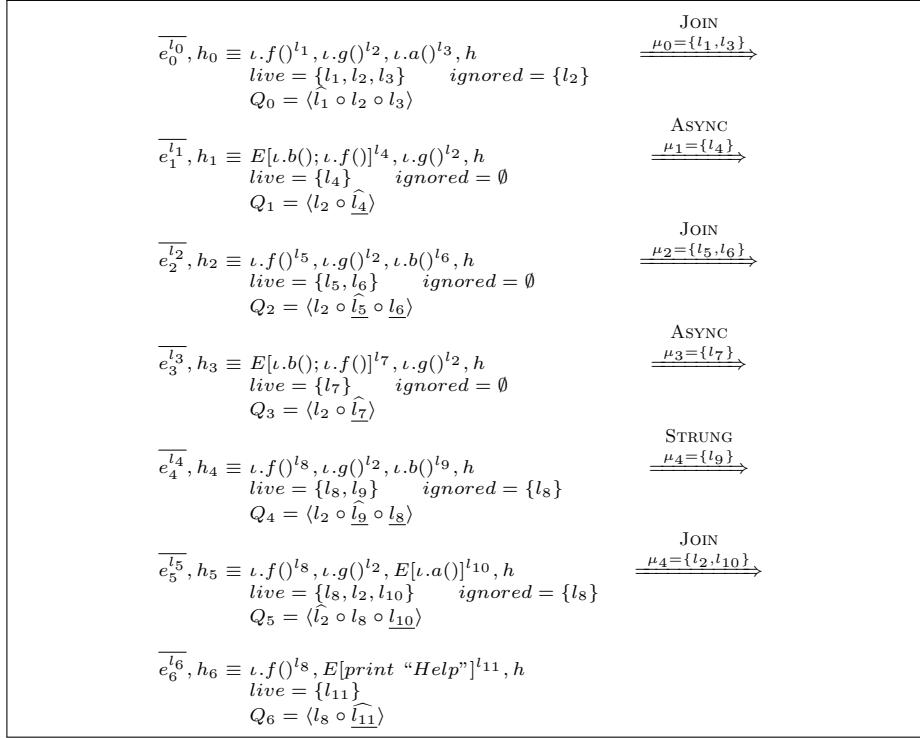


Fig. 14. A strongly-fair execution.

Two operations are defined on queues: removing and appending labels – neither operation affects the order. Removal uses the *retain* operator \upharpoonright , which is applied to a queue Q and a set of labels S as $Q \upharpoonright S$, and results in Q' , where all those labels in Q which are not in S have been removed, and the remaining labels have not changed order. Appending a queue of labels $\langle l_{k+1} \circ \dots \circ l_{k+h} \rangle$ to Q , written as $Q \circ \langle l_{k+1} \circ \dots \circ l_{k+h} \rangle$, results in a new queue, Q' , where the appended labels appear rightmost and the existing labels have not changed order: $Q' = \langle l_1 \circ \dots \circ l_k \circ l_{k+1} \circ \dots \circ l_{k+h} \rangle$. Appending a set of labels, S , to a queue, Q , written as $Q \circ S$, results in the set being treated as a queue where the order of the labels is unspecified.

To establish a notion of completeness for the recording of strong fairness constraints, the definition of strongly-fair executions below requires a finite initial configuration and an initial queue which contains all initially active labels.

Consider the example of a strongly-fair execution in figure 14 using the class `StrongFairnessExample` from listing 1.4, with the same initial configuration consisting of two synchronous method invocations `f` and `g`, and an asynchronous invocation of method `a`. The first live label in the queue is denoted with a hat.

The first four evaluation steps coincide with the original example. At this point we assumed an indefinite repetition of the following pattern: the second

chord always joins through the evaluation rule JOIN and consumes the asynchronous invocation of method **b**, with the latest synchronous invocation of method **f** participating in the join. Were this pattern to continue forever, l_2 would never regain its liveness, and each of the newly created labels would be consumed by the subsequent joining of the second chord; since no label would be ignored infinitely often, the execution would be admissible as strongly-fair.

If the execution ever involved the evaluation rule STRUNG with the current asynchronous invocation of method **b**, then l_2 would regain its liveness. If we were to allow l_2 to be ignored for the second time (as in the original example), the entire pattern of execution up to now could be repeated infinitely often, and thus l_2 could be ignored infinitely often, making the execution not strongly-fair.

Therefore, if after the fourth evaluation, the label l_9 is placed before the label l_8 in the queue Q_4 , at the fifth evaluation step STRONG will select l_9 to participate, indeed resulting in l_2 regaining its liveness in the resulting configuration (indexed by 5). Now, however, l_2 is the earliest live label in the queue (Q_5), and thus must be selected to participate in the next evaluation step (μ_5).

Were there two invocations of the method **g** in the initial configuration, both could have been ignored in the first step and placed in the queue. Following a similar pattern of execution, they would both regain their liveness at the sixth configuration. The one closest to the head of the queue would be selected to participate, while the other would be ignored a second time. If the pattern were to repeat, the remaining invocation of method **g** would regain its liveness, but now be the first live label in the queue, participating in the next evaluation step.

Although a label can be ignored a finite number of times it is not possible to determine how many times a label will be ignored as this is a consequence of its placement in the queue and its relative order to other competing labels, which is unspecified.

7 Conclusions

From the various implementations of chords, both language-based and library-based, we noticed an implicit design assumption with regards to scheduling: the scheduler is assumed to treat processes (or threads) in a “fair” way, or to not arbitrarily delay a process capable of evaluating.

We define a small labelled calculus ^lSCHOOL, and then weak and strong fairness for it, enabling the creation of abstract schedulers for chorded languages which satisfy the two notions of fairness. The labelling mechanism gives us a basis for stating the aforementioned fairness notions, as well as properties such as liveness and liveness behaviours, such as worst-case and alternating liveness.

Our weakly-fair scheduler solves the problem of processes being arbitrarily ignored continuously; the mechanism consists of tracking those processes which have been given a chance to evaluate, or have been “serviced”, and attempts to eliminate all outstanding non-serviced labels. Weakly-fair executions are stated in terms of local finite sequences, and such sequences, once generated, can be freely appended.

Our strongly-fair scheduler solves the problem of processes being arbitrarily infinitely-often ignored; the mechanism consists of a priority queue which records all fresh processes, and forces the selection of a process after a finite delay in such a way as to guarantee that all executions of arbitrarily large size result in processes which eventually either execute or terminate.

We observed the possible liveness behaviours of chorded programs, and obtained a notion of liveness alternation which we used to show worst-case delays of processes under strong fairness. As the underlying priority queue for strong fairness is bound by the number of concurrent processes, these worst-case calculations are also relevant to the running size of the scheduler's queue.

Our belief is that schedulers tend to be written using queues, so if fair, they are strongly-fair. Strongly-fair schedulers have a central point of control, so they are not suited to parallel or multi-core execution. As our treatment of fairness for chorded programs focussed on processes rather than something completely chord specific, it is likely that similar results would also hold true for schedulers for other concurrency constructs.

Acknowledgements We would like to thank Neil Datta, Anastasia Niarchou, Sebastian Hunt, Maribel Fernández, and Sophia Drossopoulou for their comments.

References

1. Nick Benton, Gavin Bierman, Luca Cardelli, Erik Meijer, Claudio Russo, and Wolfram Schulte. C_ω . <http://research.microsoft.com/Comega/>, 2004.
2. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C^\sharp . In B. Magnusson, editor, *Proc. of ECOOP02*, volume 2374, pages 415–440. Springer Press, 2002.
3. Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for C^\sharp . In *Synchronization and Concurrency in Object-Oriented Languages (SCOOOL) Workshop, OOPSLA*, October 2005.
4. Gerardo Costa and Colin Stirling. Weak and strong fairness in ccs. *Information and Computation*, 73(3):207–244, 1987.
5. Vincent Cremet. Join definitions in scala. <http://lamp.epfl.ch/cremet/join.in.scala/>.
6. Cédric Fournet. The jocaml language. <http://jocaml.inria.fr/>, 2008.
7. Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, United States, 1996. ACM Press, New York, USA.
8. Cédric Fournet and Georges Gonthier. The join calculus: a language for distributed mobile programming. *Applied Semantics Summer School*, pages 1–66, 2008.
9. Nissim Francez. *Fairness*. ACM Press, New York, USA, 1986.
10. M. Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989.
11. Claudio Russo. The joins concurrency library. In *PADL*, 2007.
12. Claudio Russo. Join patterns for visual basic. In *OOPSLA*, 2008.

A ^lSCHOOL

4

Definition 5 (Labels).

Labels, l, l', l_i , belong to a universe of labels, \mathcal{L} , extended with a special empty label, ε , to form the universe with empty labels, \mathcal{L}^ε ; hence, $l \in \mathcal{L}^\varepsilon \equiv l \in \mathcal{L} \cup \{\varepsilon\}$. Two labels, l and l' , are said to be distinct if they are not equal to each other; hence, l, l' distinct iff $l \neq l'$. We abbreviate a collection of labels as \bar{l} .

Definition 6 (Labelled Configuration).

A configuration, e_1, \dots, e_n, h , is labelled by annotating each expression with a label:

$$\text{labelled}(e_1, \dots, e_n) = e_1^{l_1}, \dots, e_n^{l_n}$$

and furthermore is uniquely labelled if each non-empty label is distinct from every other non-empty label:

$$\forall i, j \in 1..n : l_i = l_j \neq \varepsilon \implies i = j$$

We abbreviate a labelled configuration as: \bar{e}^l, h .

Definition 7 (Unlabelled Configuration).

$$\text{unlabelled}(e_1^{l_1}, \dots, e_n^{l_n}) = e_1, \dots, e_n$$

Lemma 1 (Labelling Preserves Unlabelled Form).

$$\text{unlabelled}(\text{labelled}(e_1, \dots, e_n)) = e_1, \dots, e_n$$

Proof. Straightforward from definitions 6 and 7.

Definition 8 (Evaluation Step).

An evaluation step has the following form:

$$e_1^{l_1}, \dots, e_n^{l_n}, h \xrightarrow{\mu} e_1^{l'_1}, \dots, e_m^{l'_m}, h'$$

where an initial uniquely labelled configuration, $e_1^{l_1}, \dots, e_n^{l_n}, h$, evaluates to a unique configuration, $e_1^{l'_1}, \dots, e_m^{l'_m}, h'$, through a non-empty set of participating labels, μ , which may not contain the empty label, ε , and which do not appear in the resulting configuration (they are consumed). The final configuration may be larger than the initial configuration, and hence $m \geq n$. We abbreviate an evaluation step as: $\bar{e}^l, h \xrightarrow{\mu} \bar{e}^{l'}, h'$.

Definition 9 (Execution).

A (possibly infinite) execution has the form:

$$\bar{e}_0^{l_0}, h_0 \xrightarrow{\mu_0} \bar{e}_1^{l_1}, h_1 \xrightarrow{\mu_1} \bar{e}_2^{l_2}, h_2 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \bar{e}_n^{l_n}, h_n \xrightarrow{\mu_n} \dots$$

⁴ The appendices will be available from a permanent website.

Definition 10 (Labels of a Configuration).

$$\text{labels} \left(e_1^{l_1}, \dots, e_n^{l_n} \right) = \{l_1, \dots, l_n\}$$

Definition 11 (Active Labels).

$$\text{active} \left(e_1^{l_1}, \dots, e_n^{l_n} \right) = \{l_i \in \text{labels} \left(e_1^{l_1}, \dots, e_n^{l_n} \right) : l_i \neq \varepsilon\}$$

Definition 12 (Freshness of Labels).

$$\left. \begin{array}{l} \overline{e_0^{l_0}}, h_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1 \xrightarrow{\mu_1} \overline{e_2^{l_2}}, h_2 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow{\mu_n} \dots \\ \exists i \in \mathbb{N} : \exists l \in \text{active} \left(\overline{e_i^{l_i}} \right) \\ \forall j, 0 \leq j \leq i-1 : l \notin \text{active} \left(\overline{e_j^{l_j}} \right) \\ l \text{ is fresh at } i \end{array} \right\} \implies$$

Lemma 2 (Participating Labels Form A Set).

$$\overline{e^l}, h \xrightarrow{\{l_1, \dots, l_n\}} \overline{e^{l'}}, h' \implies \forall i, j \in 1..n : l_i = l_j \implies i = j$$

Proof. By structural induction on the derivation of $\overline{e^l}, h \xrightarrow{\{l_1, \dots, l_n\}} \overline{e^{l'}}, h'$; case analysis on the last evaluation rule applied.

Lemma 3 (Participating Labels Are Never Empty).

$$\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h' \implies \mu \neq \emptyset$$

Proof. By structural induction on the derivation of $\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h'$; case analysis on the last evaluation rule applied.

Lemma 4 (Empty Labels Never Participate).

$$\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h' \implies \varepsilon \notin \mu$$

Proof. By structural induction on the derivation of $\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h'$; case analysis on the last evaluation rule applied.

Lemma 5 (Participating Labels are Consumed).

$$\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h' \implies \forall l \in \mu : l \notin \text{labels} \left(\overline{e^{l'}} \right)$$

Proof. By structural induction on the derivation of $\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h'$; case analysis on the last evaluation rule applied.

Lemma 6 (Labels Remain Finite).

$$\left. \begin{array}{l} \exists n \in \mathbb{N} : |\text{labels} \left(\overline{e^l} \right)| = n \\ \overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h' \end{array} \right\} \implies \exists m \in \mathbb{N} : |\text{labels} \left(\overline{e^{l'}} \right)| = m$$

Proof. By structural induction on the derivation of $\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h'$; case analysis on the last evaluation rule applied.

Lemma 7 (Upper Bound On Creation Of Labels).

$$\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h' \implies \exists \lambda \in \mathbb{N} : |\text{active}(\overline{e^{l'}})| \leq |\text{active}(\overline{e^l})| + \lambda$$

Proof. By structural induction on the derivation of $\overline{e^l}, h \xrightarrow{\mu} \overline{e^{l'}}, h'$; case analysis on the last evaluation rule applied.

Observation 1 (Liveness Behaviour of a Label)

For a (possibly infinite) execution sequence

$$\overline{e_0^{l_0}}, h_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1 \xrightarrow{\mu_1} \overline{e_2^{l_2}}, h_2 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow{\mu_n} \dots$$

assume a label, l , appears in the execution for the first time at configuration i , and hence $l \in \text{labels}(\overline{e_i^{l_i}}) \wedge \forall j \in \mathbb{N}, 0 \leq j < i : l \notin \text{labels}(\overline{e_j^{l_j}})$; then its liveness behaviour is characterised as one out of the following two:

1. l alternates between live and non-live always, or
2. l alternates between live and non-live α times, and then remains either always non-live or always live, hence $\exists \theta_0, \theta_1, \dots, \theta_{\alpha-1} \in \mathbb{N}$, with θ_0 denoting the number of configurations before the first alternation, starting from the i 'th configuration, and $\forall m, 1 \leq m \leq \alpha - 1$, θ_m denoting the number of configurations between the m 'th alternation and the $m + 1$ 'th alternation.

Therefore, depending on the two cases:

(a) l remains always non-live, we have: $\forall k \geq \sum_{m=0}^{\alpha-1} \theta_m + 1 : l \notin \text{live}(\overline{e_k^{l_k}})$,
or

(b) l remains always live, we have: $\forall k \geq \sum_{m=0}^{\alpha-1} \theta_m + 1 : l \in \text{live}(\overline{e_k^{l_k}})$

There are two special cases when $\alpha = 0$; either l remains always non-live, and hence $\forall k \geq i : l \notin \text{live}(\overline{e_k^{l_k}})$, or l remains always live, and hence $\forall k \geq i : l \in \text{live}(\overline{e_k^{l_k}})$.

Observation 2 (Largest θ and Worst Case Behaviour)

For a (possibly infinite) execution sequence

$$\overline{e_0^{l_0}}, h_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1 \xrightarrow{\mu_1} \overline{e_2^{l_2}}, h_2 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow{\mu_n} \dots$$

assume a label, l , appears in the execution for the first time at configuration i , and hence $l \in \text{labels}(\overline{e_i^{l_i}}) \wedge \forall j \in \mathbb{N}, 0 \leq j < i : l \notin \text{labels}(\overline{e_j^{l_j}})$; then from observation 1 we know that it will alternate either always or α times. In the latter case, it is possible to establish the largest number, θ , of configurations between two alternations: $\theta \equiv \max\{\theta_0, \dots, \theta_{\alpha-1}\}$. The first example execution of figure 6 illustrates a θ .

B Weak Fairness

Definition 13 (Weakly-Fair Evaluation Step).

A weakly-fair evaluation step has the following form:

$$e_1^{l_1}, \dots, e_n^{l_n}, h, L \xrightarrow{\mu} e_1^{l'_1}, \dots, e_m^{l'_m}, h', L'$$

where an initial labelled configuration, $e_1^{l_1}, \dots, e_n^{l_n}, h$, evaluates to a final labelled configuration, $e_1^{l'_1}, \dots, e_m^{l'_m}, h'$, through a non-empty set of participating labels, μ , which may not contain the empty label, ε , and which do not appear in the resulting configuration (they are consumed), and also where a set of serviced labels, L , is maintained. The final configuration may be larger than the initial configuration, and hence $m \geq n$. We abbreviate a weakly-fair evaluation step as: $\overline{e^l}, h, L \xrightarrow{\mu} \overline{e^{l'}}, h', L'$. A (possibly infinite) sequence of weakly-fair evaluation steps thus has the form:

$$\overline{e_0^{l_0}}, h_0, L_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1, L_1 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n, L_n \xrightarrow{\mu_n} \dots$$

Definition 14 (Locally Weakly-Fair Execution).

$$\overline{e^l}, h \xrightarrow[L]{M} \overline{e^{l'}}, h' \text{ iff}$$

$$\exists k \in \mathbb{N} :$$

$$\overline{e_0^{l_0}}, h_0, L_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1, L_1 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_{k-1}} \overline{e_k^{l_k}}, h_k, L_k$$

$$\text{where } \overline{e^l}, h \equiv \overline{e_0^{l_0}}, h_0 \quad \text{and} \quad \overline{e^{l'}}, h' \equiv \overline{e_k^{l_k}}, h_k$$

$$\text{and } M \equiv \mu_0 \circ \mu_1 \circ \dots \circ \mu_{k-1} \quad \text{and} \quad L \equiv L_k$$

$$\wedge \text{ live}(\overline{e^l}) \subseteq L$$

$$\wedge \exists m \in \mathbb{N} : |\text{labels}(\overline{e_0^{l_0}})| = m$$

$$\wedge L_0 = \emptyset$$

Definition 15 (Weakly-Fair Execution).

$$\overline{e^l}, h \Rightarrow \overline{e^{l'}}, h' \text{ iff}$$

$$\exists n \in \mathbb{N} :$$

$$\overline{e_0^{l_0}}, h_0 \xrightarrow[L_0]{M_0} \overline{e_1^{l_1}}, h_1 \xrightarrow[L_1]{M_1} \dots \xrightarrow[L_{n-1}]{M_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow[L_n]{M_n} \dots$$

$$\text{where } \overline{e^l}, h \equiv \overline{e_0^{l_0}}, h_0 \quad \text{and} \quad \overline{e^{l'}}, h' \equiv \overline{e_n^{l_n}}, h_n$$

Lemma 8 (Correctness of Locally Weakly-Fair Execution).

$$\left. \begin{array}{l} \overline{e^l}, h \xrightarrow[L]{M} \overline{e^{l'}}, h' \quad \text{where} \\ \overline{e^l}, h \equiv \overline{e_0^{l_0}}, h_0 \quad \text{and} \quad \overline{e^{l'}}, h' \equiv \overline{e_k^{l_k}}, h_k \quad \text{for some } k \in \mathbb{N} \quad \text{and} \\ \overline{e_0^{l_0}}, h_0, L_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1, L_1 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_{k-1}} \overline{e_k^{l_k}}, h_k, L_k \\ \forall l \in \text{live}(\overline{e_0^{l_0}}) : \exists k', 0 < k' \leq k : l \notin \text{live}(\overline{e_{k'}^{l_{k'}}}) \end{array} \right\} \Rightarrow$$

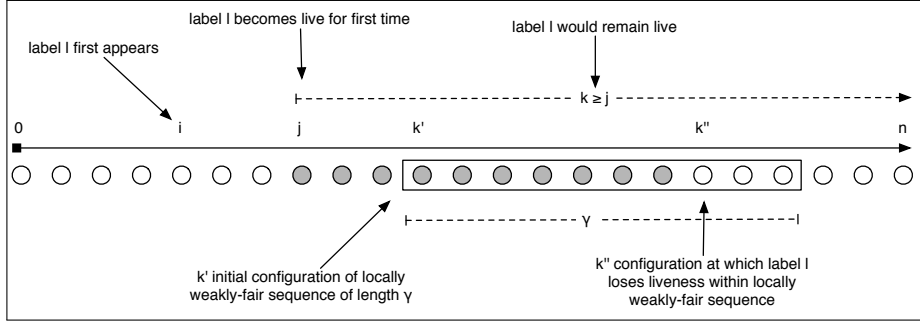


Fig. 15. Visual description of correctness theorem for weak fairness.

Proof. By repeated application of the WEAK selection rule which corresponds to the locally weakly-fair execution; since by definition 14 all initially live labels must be included in the final set of serviced labels, at least one application of WEAK will result in a configuration where each label is not live.

Theorem 1 (Correctness of Weakly-Fair Execution).

$$\overline{e^l}, h \Rightarrow \overline{e^{l'}}, h' \Rightarrow$$

$$\forall i \in \mathbb{N} : \forall l \in \text{labels} \left(\overline{e_i^{l_i}} \right) : \exists k \geq i : l \notin \text{live} \left(\overline{e_k^{l_k}} \right)$$

Proof. By contradiction, using lemma 8: assume a label l from configuration i becomes live at configuration $j \geq i$ and remains live $\forall k \geq j$ configurations; but then l will be live at some configuration $k' \geq j$ which is also the initial configuration of a locally weakly-fair execution sequence of some length γ , hence by the above lemma $\exists k'', k' < k'' \leq k' + \gamma - 1 : l \notin \text{live} \left(\overline{e_{k''}^{l_{k''}}} \right)$.

Theorem 2 (Weakly-Fair and ^lSCHOOL Correspondence).

$$\overline{e^l}, h \Rightarrow \overline{e^{l'}}, h' \Rightarrow$$

$$\overline{e_0^{l_0}}, h_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1 \xrightarrow{\mu_1} \overline{e_2^{l_2}}, h_2 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow{\mu_n} \dots$$

Proof. Straightforward by expanding a weakly-fair execution sequence into the sequence of locally weakly-fair execution sequences, and then dropping all references to sets of serviced labels.

C Strong Fairness

Definition 16 (Strongly-Fair Evaluation Step).

A strongly-fair evaluation step has the following form:

$$e_1^{l_1}, \dots, e_n^{l_n}, h, Q \xRightarrow{\mu} e_1^{l'_1}, \dots, e_m^{l'_m}, h', Q'$$

where an initial labelled configuration, $e_1^{l_1}, \dots, e_n^{l_n}, h$, evaluates to a final labelled configuration, $e_1^{l'_1}, \dots, e_m^{l'_m}, h'$, through a non-empty set of participating labels,

μ , which may not contain the empty label, ε , and which do not appear in the resulting configuration (they are consumed), and also where a queue of pending labels, Q , is maintained. The final configuration may be larger than the initial configuration, and hence $m \geq n$. We abbreviate a strongly-fair evaluation step as: $\overline{e^l}, h, Q \xrightarrow{\mu} \overline{e^{l'}}, h', Q'$. A (possibly infinite) sequence of strongly-fair evaluation steps thus has the form:

$$\overline{e_0^{l_0}}, h_0, Q_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1, Q_1 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n, Q_n \xrightarrow{\mu_n} \dots$$

Definition 17 (Strongly-Fair Execution).

$$\begin{aligned} \overline{e^l}, h, Q \Rightarrow \overline{e^{l'}}, h', Q' \text{ iff} \\ \exists n \in \mathbb{N} : \\ \overline{e_0^{l_0}}, h_0, Q_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1, Q_1 \xrightarrow{\mu_1} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n, Q_n \xrightarrow{\mu_n} \dots \\ \text{where } \overline{e^l}, h, Q \equiv \overline{e_0^{l_0}}, h_0, Q_0 \quad \text{and} \quad \overline{e^{l'}}, h', Q' \equiv \overline{e_n^{l_n}}, h_n, Q_n \\ \wedge \exists m \in \mathbb{N} : |\text{labels}(\overline{e_0^{l_0}})| = m \\ \wedge Q_0 = \langle \rangle \circ \text{active}(\overline{e_0^{l_0}}) \end{aligned}$$

Lemma 9 (Upper Bound On Distance From Head Of Queue).

$$\overline{e^l}, h, Q \Rightarrow \overline{e^{l'}}, h', Q' \Rightarrow$$

$$\begin{aligned} \forall i \in \mathbb{N} : l \in Q_i \Rightarrow \\ \exists k, \lambda, \rho \in \mathbb{N}, k < i * \lambda + \rho : Q_i = \langle l_1 \circ \dots \circ l_k \circ l \circ \dots \rangle \end{aligned}$$

Proof. By induction on the length of the execution and use of lemma 7.

Lemma 10 (Queue Is A Permutation Of Configuration).

$$\overline{e^l}, h, Q \Rightarrow \overline{e^{l'}}, h', Q' \Rightarrow \forall i \in \mathbb{N} : Q_i \text{ is a permutation of } \text{active}(\overline{e_i^{l_i}})$$

Proof. By induction on the length of the execution and application of STRONG.

Lemma 11 (Order of Labels Preserved).

$$\begin{aligned} \overline{e^l}, h, Q \Rightarrow \overline{e^{l'}}, h', Q' \Rightarrow \\ \forall i \in \mathbb{N} : \forall l, l' \in Q_i : l \leq_{Q_i} l' \wedge l, l' \in Q_{i+1} \Rightarrow l \leq_{Q_{i+1}} l' \\ \wedge \forall l \in Q_i : \forall l' \in Q_{i+1}, l' \notin Q_i : l \leq_{Q_{i+1}} l' \end{aligned}$$

Proof. Straightforward by inspection of the STRONG selection rule; existing labels are passed along in the same order when not removed, and all new labels are appended after the existing labels.

Lemma 12 (Ignored Labels Move Toward Head Of Queue).

$$\left. \begin{aligned} \overline{e^l}, h, Q \Rightarrow \overline{e^{l'}}, h', Q' \\ l \text{ was ignored at } i \\ Q_i = \langle l_1 \circ \dots \circ l_k \circ l \circ \dots \rangle \end{aligned} \right\} \Rightarrow \begin{aligned} \exists k' \in \mathbb{N}, k' < k : \\ Q_{i+1} = \langle l_1 \circ \dots \circ l_{k'} \circ l \circ \dots \rangle \end{aligned}$$

Proof. Through application of the STRONG selection rule and using lemmas 5, 10 and 11.

Lemma 13 (First Live Label Participates).

$$\left. \begin{array}{l} \overline{e^l, h, Q} \Rightarrow \overline{e^{l'}, h', Q'} \\ Q_i = \langle l \circ \dots \rangle \\ l \in \text{live} \left(\overline{e_i^{l_i}, h_i} \right) \end{array} \right\} \Longrightarrow l \in \mu_i$$

Proof. Through application of the STRONG selection rule and the definition for the queue ordering operator \leq_Q .

Lemma 14 (Upper Bound On Ignoring A Label).

$$\overline{e^l, h, Q} \Rightarrow \overline{e^{l'}, h', Q'} \Longrightarrow$$

$$\forall i \in \mathbb{N} : \forall l \in Q_i : \exists k, \lambda, \rho \in \mathbb{N}, k \leq i * \lambda + \rho - 1 : \exists \tau_1, \dots, \tau_k \in \mathbb{N} : \\ l \text{ ignored at } i + t \text{ for some } t \in \mathbb{N} \Longrightarrow t \in \{\tau_1, \dots, \tau_k\}$$

Proof. Assume that a label l appears for the first time at configuration i , and hence by lemma 10 will be in the queue Q_i ; then by lemma 9 we know that there is an upper bound on the distance of that label from the head of the queue, or: $\exists k', \lambda, \rho \in \mathbb{N}, k' < i * \lambda + \rho : Q_i = \langle l_1 \circ \dots \circ l_{k'} \circ l \circ \dots \rangle$. From lemma 12 we know that each time l is ignored it will move towards the head of the queue; in the worst case, this will happen $\kappa' - 1$ times, after some t steps, whereupon the label will be the first in the queue, or: $Q_{i+t} = \langle l \circ \dots \rangle$. From lemma 13 we know that the next time l will be live it must be selected for participation, and hence it cannot be ignored again after configuration $i + t$; therefore, there are at most $\kappa = \kappa' - 1$ possible values, $\{\tau_1, \dots, \tau_\kappa\}$, for t , and hence: l ignored at $i + t$ for some $t \in \mathbb{N} \Longrightarrow t \in \{\tau_1, \dots, \tau_\kappa\}$.

Theorem 3 (Correctness of Strongly-Fair Execution).

$$\overline{e^l, h, Q} \Rightarrow \overline{e^{l'}, h', Q'} \Longrightarrow \\ \forall i \in \mathbb{N} : \forall l \in \text{labels} \left(\overline{e_i^{l_i}} \right) : \exists j \geq i : \forall k \geq j : l \notin \text{live} \left(\overline{e_k^{l_k}} \right)$$

Proof. From lemma 14 we know that each label, l , will be ignored at configurations $i + t$ for some $t \in \{\tau_1, \dots, \tau_r\}$ and finite r ; but then we immediately obtain that $\exists j \geq i + \max(\{\tau_1, \dots, \tau_r\}) + 1 : \forall k \geq j : l \notin \text{live} \left(\overline{e_k^{l_k}} \right)$.

Theorem 4 (Strongly-Fair and ^lSCHOOL Correspondence).

$$\overline{e^l, h, Q} \Rightarrow \overline{e^{l'}, h', Q'} \Longrightarrow \\ \overline{e_0^{l_0}, h_0} \xrightarrow{\mu_0} \overline{e_1^{l_1}, h_1} \xrightarrow{\mu_1} \overline{e_2^{l_2}, h_2} \xrightarrow{\mu_2} \dots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}, h_n} \xrightarrow{\mu_n} \dots$$

Proof. Straightforward by expanding the strongly-fair execution and then dropping all references to queues.