University of London
Imperial College of Science, Technology and Medicine
Department of Computing

On The Design of Chorded Languages

Alexis Petrounias

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, August 2008.

Declaration

This thesis conforms to the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College. The research presented here has not been submitted for any degree or comparable award of this or any other university or institution. The thesis was supervised by Susan Eisenbach.

Abstract

Chords are a concurrency mechanism of object-oriented languages inspired by the join of the Join Calculus. They represent the combination of object-based program structuring and the *chemical metaphor* for concurrency. Several modern languages feature chords, among which are Polyphonic $C^\sharp$ and C_ω. Their proponents say that their use will raise the level of abstraction concurrent programs are written in, hence increasing the likelihood of producing correct programs.

We present SCHOOL, the Small Chorded Object-Oriented Language, a featherweight model which aims to capture the essence of the concurrent behaviours of chords. Our model serves as a generalisation of chorded behaviours found in existing experimental languages such as Polyphonic $C^\sharp$. Furthermore, we study the interaction of chords with fields by extending SCHOOL to include fields, resulting in ^{f}SCHOOL. Fields are orthogonal to chords in terms of concurrent behaviours. We show that adding fields to SCHOOL does not change its expressiveness by means of an encoding between the two languages.

We observe that most implementations of concurrency in programming languages assume fair scheduling. Two primary notions of fairness are weak and strong, and hence studying the way these two notions apply to chorded languages serves as a basis for scheduler specification. We therefore define weak and strong fairness for SCHOOL which allows us to classify executions as admissible or inadmissible under the two notions of fairness. We develop two abstract schedulers in the form of high-level selection rules imposed on the underlying language, which allow us to admit exactly the weakly- and strongly-fair executions. We also present worst-case calculations for the delay of individual processes.

Acknowledgements

I am grateful to my supervisor, Susan Eisenbach, and my second supervisor, Sophia Drossopoulou, for their guidance and support. I wish to thank my examiners, Maribel Fernández and Sebastian Hunt, for their comments, which have greatly improved this work.

The following people have offered comments on this work: Christopher Anderson, Nick Benton, Alex Buckley, Luca Cardelli, David Cunningham, Feruccio Damianni, Neil Datta, Cédric Fournet, Philippa Gardner, Paola Giannini, Georges Gonthier, Khilan Gudka, Tim Harris, Paul Kelly, John Knottenbelt, Doug Lea, Andrew McVeigh, Alok Mishra, Dimitrios Mostrous, Anastasia Niarchou, Iain Phillips, Mathew Sackman, Gareth Smith, Mathew Smith, Paul Trick, Sebastian Uchitel, Daniele Varacca, Herbert Wiklicky, and Nobuko Yoshida.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

We are interested in the design and implementation of high-level programming languages which can be used to program complex concurrent systems.

Concurrent systems are becoming increasingly important components of our technological infrastructure. Several key reasons can be identified: the vast expansion of computer networks, the exponential increase in connectivity between devices and users of these devices, the reliance on decentralised protocols, collaborative business models, the ubiquity of multi-core processors, the necessity of adaptiveness to heterogeneous environments, the reliance on real-time monitoring and control of physical processes, and many more.

Concurrency gives rise to a set of problems not found in classical computational systems which are monolithic and sequential. Generally, the non-deterministic concurrent execution of processes increases the complexity in reasoning about and identifying the properties of these systems.

The aforementioned difficulties make the design and implementation of concurrent systems hard, error-prone, almost impossible to test thoroughly, and sometimes dangerous. Furthermore, such systems are usually large and expensive by nature, increasing the damaging effect of the identified difficulties.

1.1 Motivation

Programmers must either become more competent with the tools they currently possess, or they require more sophisticated tools than currently are available for building concurrent systems. Programming languages and their associated compilers and analyses are fundamental tools which are used both for describing and implementing systems.

Additionally, we argue that contemporary languages offer constructs generally unsuitable for use in concurrent software, either because these constructs result in complicated and error-prone software or because they do not reflect the underlying computational paradigm of concurrency.

Thus we justify our efforts in providing appropriate language constructs with which people can think about the programming of concurrent systems, develop software, reason about the properties of the software, and argue rigourously that the systems' requirements have been satisfied.

In the remainder of this section we will give a brief overview of the contemporary practise in concurrency as identified in language design (section 1.1.1 below) and outline the fundamental problems which motivate us to investigate alternative approaches. We continue with an overview of one such alternative approach, the *chemical metaphor* [12, 11, 13, 40] for concurrency (section 1.1.2 below), and present several of its aspects which make it appealing as a concurrent model for programming language design.

1.1.1 The Contemporary Methodology

The constructs employed by contemporary object-oriented languages are primarily imperative in nature; although fairly widespread and in many cases considered the *de facto* standard, it is possible to argue successfully against their suitability, in many cases, for the purpose of constructing concurrent software.

Worse, many of these constructs have only been retrofitted into languages in the form of libraries (POSIX threads [18], Java Concurrency [37], and so on). This can be primarily attributed to the dominating paradigm of contemporary concurrency which views sequential programs as normal and concurrent interaction as a special case.

We identify two fundamental reasons due to which this paradigm is becoming increasingly unsuitable in contemporary software design: the qualitative change in our requirements of computational tools, and the physical limitations of computing hardware.

The first reason stems from the increasing use of software for the control and interaction with the physical world. Physical and social phenomena and the requirements of autonomy and robustness they impose on computational devices comprise a highly non-deterministic, complex world in which events and actions cannot take place in the safe confines of a discrete and solid operating system or framework.

Concurrent actions and distributed program execution become the norm, and malleable computational devices and network topologies become the underlying software platform.

The techniques of sequential computation - along with its workarounds for concurrency - diminishes in effectiveness and applicability. Indeed, it is astonishing how even the tiniest of programs, such as the interleaving of two reading and writing processes, yield such a vast number of problems and associated research.

The second reason stems from the physical limitations of computational hardware design: it appears that processor circuitry has reached a stage where increasing performance for sequential computation is very hard to achieve, and designers are resorting to increasing parallelism and hence throughput via the introduction of multiple processors and "multi-core" processors. Thus, even applications disjoint from the complexities of the physical world will now face

a highly concurrent environment as the basic operating system in which they execute.

1.1.2 An Alternative Approach

An overview of contemporary programming languages used for concurrency leads us to identify a certain kind of complacency with regards to the underlying paradigms these languages reflect. We feel obliged to consider a paradigm for concurrency fundamentally different to what is accepted as the norm today.

Attempting, therefore, to shift the underlying paradigm to that which accepts concurrent programs as the norm we base our work on the chemical concurrency metaphor of Berry and Boudol [17], and its long history from the "language-like" chemistry semantics of Γ [12, 11] to the join calculus of Fournet and Gonthier [29].

The chemical metaphor of concurrency is appealing in two respects: it is a resource-centric description of computational requirements, and it expresses several important programming concepts concisely and elegantly.

Generally in chemical concurrency processes and data are both first-class citizens in a computational environment where basic tasks such as communication and scheduling are *managed* by the underlying platform, rather than being rigidly encoded in a program. This modelling of resources is desirable when processes are mobile, resources are distributed, and multiple devices communicate in order to reach a common desired goal or consensus.

Specifically in chemical concurrency, computation is non-deterministic, communication is based on messages and their routing, processes are reactive to events and other processes' messages, and consensus (rendezvous) forms the principal means of control.

This paradigm of concurrency is not intended as a practical programming language, rather, it promotes a way of thinking about the modelling of software systems, and demonstrates functionality which is desirable in a programming

language used for building such systems.

Considering, therefore, the design of a programming language, we find that the software-as-objects metaphor is a promising direction. Objects are appealing in their own right, as they promote modelling of resources, communication, and separation of computational tasks, not unlike the chemical metaphor. However, objects on their own do not offer much in terms of concurrency.

Their high-level compatibility with the chemical metaphor leads to a promising combination of the two with which we obtain a system where many physical objects and processes can be elegantly modelled: consider, as a motivating example, the case where an aeroplane is attempting to land; our system in a very abstract sense consists of two objects, the aeroplane and the runway, which both actively attempt to reach a consensus (the landing).

The object metaphor enables us to manage a system with increasing complexity as more physical elements are modelled: multiple aeroplanes, multiple runways, a global distributed network of airports, and so on. The chemical metaphor enables us to manage the increase in the complexity of consensus when more requirements are modelled: weather events, fuel availability, landing priorities, partial landing wheel failure, and so on.

Furthermore, the combination of metaphors enables us to think of programs in a scalable manner: more requirements can be added to the consensus for landing, yet without affecting, for example, a consensus for obtaining priority over airspace; or, new kinds of physical elements can be added, such as helicopters and hellipads, without, again, affecting the interaction of aeroplane objects with runway objects.

There have been several informal advances towards this combination of metaphors, with the primary result being the notion of *chords* as they first appeared in the programming language Polyphonic C$^\sharp$ [15, 16]. Their proponents say that their use will raise the level of abstraction concurrent programs are written in, hence increasing the likelihood of producing correct programs.

Unfortunately, there is no precise semantic model of chords for Polyphonic C$^\sharp$ or other languages which feature chords (such as Scala [22] or C$_\omega$ [14]). With neither a precise semantics, nor proven technology, there is no surety that chorded programs behave sensibly. Furthermore, all aforementioned models are presented with implicit assumptions regarding scheduling and fair allocation of resources.

1.2 Aims and Contribution

We aim to develop a model with which we can systematically study chords and chorded programming language design. This model must benefit from a sound formal basis, and it should also be rigourously investigated in terms of management facilities (such as inter-object communication and process scheduling), given that so much of the function of chords is managed by the language execution (typically in a virtual machine) and not the programmer. We will explore issues related to type safety, synchronisation, message passing, and deadlock and progress (this is the content of chapter 3).

The scheduling of concurrent programs is a fundamental aspect of language design, and hence we are interested in discovering the abstractions required for specifying the scheduling of executions of chorded programs. Processes compete for the acquisition of execution time, and objects and chords compete for the consumption of messages; furthermore, execution is arbitrarily interleaved; finally, programs encode arbitrary synchronisation constraints. Therefore, describing schedulers, along with any guarantees they impose (such as fair treatment of competing processes or upper bounds on delays), is not a trivial issue (this is the content of chapter 4).

1.2.1 Formalisation of Chorded Languages

We want to give semantics to languages which feature constructs both from the object and the chemical metaphors. Our aim in this endeavour is two-fold: first, we develop a formal basis upon which we may rigorously and confidently assign meaning to chorded programs; second, we equip ourselves with a foundational model of chord semantics with which we can further investigate the interaction of chords with other constructs popular in contemporary object-oriented languages.

We describe a featherweight model for chorded languages, the Small Chorded Object-Oriented Language (SCHOOL), a kernel language which aims to capture the essence of chorded behaviours. We then investigate standard properties of this language, namely, a sound static typing system, deadlock detection, and progress. Furthermore, we use our featherweight model to investigate the interaction of chords with *mutable fields*, a standard feature of many object-oriented languages (resulting in a slightly richer language ^{f}SCHOOL); we conclude that such fields can be encoded using only chords, and hence ^{f}SCHOOL is not more expressive than SCHOOL.

1.2.2 Abstract Scheduler Specification

We want to understand the interaction of chords with processes (or *threads*), a common abstraction for concurrent object-oriented languages. Processes typically interact by exchanging messages (or *method invocations*) and by synchronising (or *joining*) or creating new processes (*spawning* new threads); most languages assume their implementations will have some kind of scheduler which manages processes in a consistent way, usually with some guarantee of progress (threads get to execute whenever they can).

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, and language constructs interfere with scheduling policies in complex ways. Hence, we investigate two primary notions of fairness, *weak* and *strong*, and describe abstract schedulers with weak and strong fairness guarantees for SCHOOL executions.

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. We also investigate the liveness behaviour of programs under strong fairness and establish worst-case behaviours in terms of scheduling delays.

1.3 Structure of This Thesis

In chapter 2 we begin by investigating the background work on the chemical metaphor for concurrency. We survey the current state of chords and their adoption on contemporary languages Polyphonic C$^\sharp$ [15, 16] and JoCaml [27, 26]. We identify two crucial missing aspects of research on chords: a formal model of chorded languages and a model of scheduling for chorded languages. Hence, we also survey the primary notions of fairness which serve as the basis for our abstract scheduler descriptions.

In chapter 3 we proceed to show the derivation of the chord construct from the combination of join concurrency and object-oriented semantics, and present the model form for all object-oriented languages which feature chords. This enables us to systematically classify several kinds of chords and show how they can be used to build programs. We then compare chorded programs to equivalents written in non-chorded languages, where such equivalents exist.

Our next step is the formalisation of chord semantics via the presentation of SCHOOL, along with basic results for type soundness, deadlock, and progress, and a comparison with Polyphonic C$^\sharp$. Finally, we present the extension of SCHOOL with mutable fields, ^{f}SCHOOL, and show how such fields can be encoded using only SCHOOL; our main result shows that adding mutable fields to chorded languages does not increase their expressive power.

In chapter 4 we present a general notion of *process liveness* for SCHOOL executions (along with a labelling scheme for identifying execution traces), and offer a mechanism for characterising executions as weakly- and strongly-fair. We then describe two abstract schedulers, one weakly- and the other strongly-fair, which serve as the basis for future concrete scheduler implementations. Finally, we investigate fair executions in terms of worst-case behaviours for the delay of interacting processes due to competition for execution time and messages.

In chapter 5 we conclude by offering several directions for future work. Specifically, we encourage the investigation of alternative derivations of the generalised chord model which serves as the basis of all chorded language models; this will enable the formulation of models which exhibit more regularity or more expressive power than SCHOOL. Furthermore, we offer some ideas on the impact of exceptions in the setting of chords, because many exception mechanisms are not orthogonal to concurrency constructs. Also, the interaction of chords with other widely-used imperative constructs of object-oriented languages (already available in Polyphonic C$^\sharp$ and Scala) such as *re-entrant monitors* and *explicit thread life-cycles*, warrant further investigation. Finally, scheduling of chords would benefit from a more systematic study of various notions of fairness, as well as the implications of fairness constraints imposed not only on processes, but on individual objects and chords as well.

Chapter 2

Background

We investigate in detail the theoretical results which form the motivation and basis of our work on chords and chorded languages. We begin by briefly setting the historical background before presenting an overview of several concepts, formalisms, and programming languages which lead to the development of join semantics and chords.

We trace the evolution of the *join* concept from the abstract prallel language Γ [13], via the chemical metaphors of the CHAM [17] and RCHAM [28], their formal description through the Join Calculus [29, 30], their functional implementation in JoCaml [27, 25, 26], and finally the inception of the contemporary notion of a *chord* in Polyphonic C♯ [15, 16]. We will also briefly look at other developments which were directly motivated by join concurrency, such as implementation strategies for JoCaml [55], functional nets [67, 69], and library-based implementations of chords [78, 22, 19, 91, 88].

Finally, we provide an overview of the problems arising from assumptions regarding the management of scheduling for concurrent processes, and focus on two primary notions of fairness, weak and strong [45], which serve as the basis for many kinds of abstract scheduler designs.

2.1 Historical Overview

In 1966 Peter J. Landin began his long and influential project ISWIM [48], an abstract family of purely functional computer programming languages based on Alonzo Church's and Stephen Kleene's λ-calculus [9]. The aim of this project was the creation of programming languages which feature the practical mathematical notation of calculi, abandoning the ad-hoc syntax of the then popular languages such as FORTRAN [10] or Algol [8].

Although never implemented, ISWIM was the primary motivation for the development of many important languages such as SASL [83], Miranda [84], ML [65], Haskell [41] and their successors. Most importantly, Landin's work pioneered the idea of developing programming languages directly from calculi.

A systematic investigation of abstract concurrent programming resulted in the first robust models of concurrency, mainly due to the work of Hoare on Occam and CSP [39, 38], Pierce and Turner on the Pict [73], Milner, Parrow, and Walker on the π-calculus [63], and Smolka, Henz and W'aurtz on Oz [81], a pioneering piece of work on object-oriented concurrency.

Concrete results from functional programming and concurrent calculi were first successfully combined with the works of Giacalone, Mishra, and Prasad on Facile [36], Reppy on CML [77], and Agha, Mason, Smith, and Talcott on Actors [3].

Against this background we now overview the principal concepts, formalisms, and programming languages which lead to the inception of chords. Figures 2.1 and 2.2 contain rough schematics of the influences which lead to the join calculus and to chorded languages. Clouds represent concepts, ovals represent formalisms or calculi, and rectangles represent programming languages.

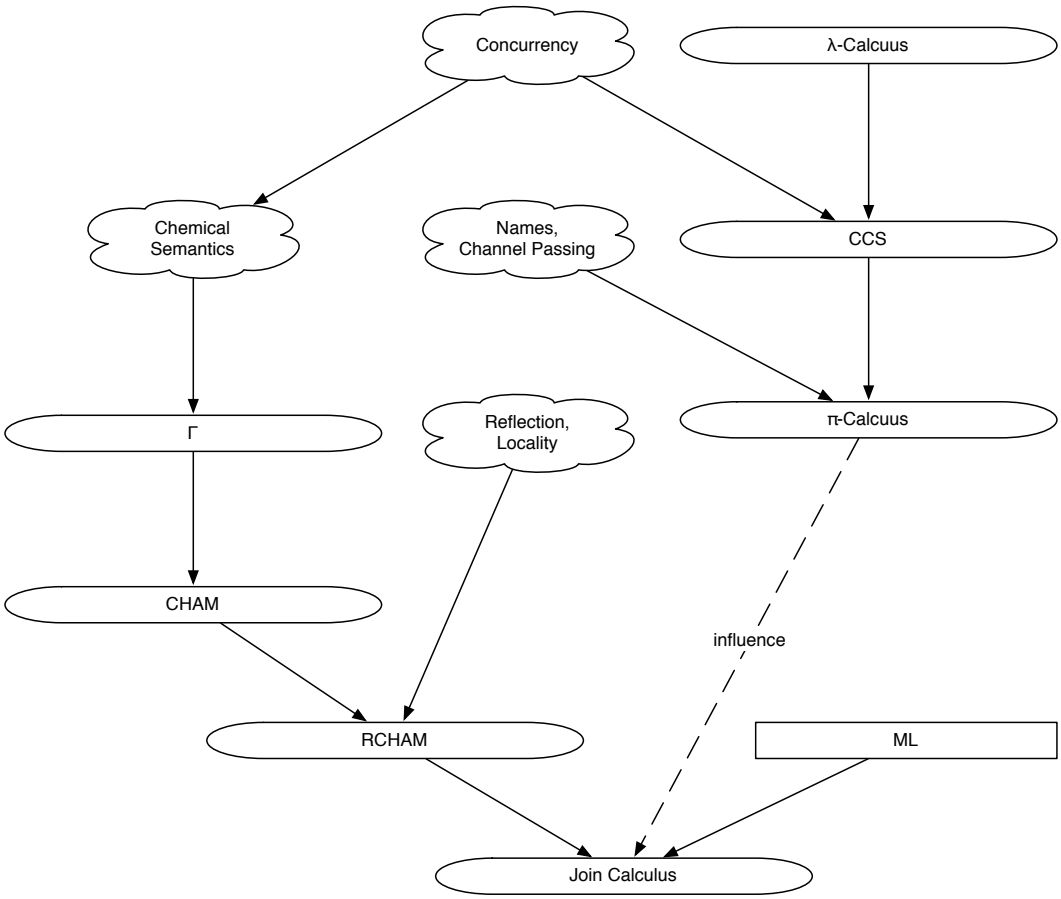

Figure 2.1: Computational Concepts and Calculi Leading to the Join Calculus

2.1.1 Computational Concepts and Calculi

The primary influence for functional programming languages, including the ISWIM and Lisp families, was the λ-calculus of Church and Kleene. Developed in the 1930's as a formal model of computation, it offered a precise definition of what constitutes a function, a function application, and recursion.

Although the λ-calculus elegantly captured the notion of sequential computation, it had nothing to offer in terms of reasoning about the *state* of computer programs. Specifically, the rigid algebraic interpretation of the λ-calculus resulted in systems lacking *interaction*, that is, systems which are not only de-

pendent on input and output, but also on actions taking place concurrently in other systems.

The most influential answer to this problem of interaction was the Calculus of Communicating Systems (CCS) [58, 57, 59, 60, 61, 56, 62] of Milner during the 1970's. CCS considers the problems of non-terminating programs, side-effects of computations, and non-determinism.

The focus of the CCS and its meta-theory is the *parallel composition* of processes. *Message-passing* is the means of communication between processes, which can be *interleaved* and their interleaving can be *observed*. Sequential composition is no longer a basic operator, rather, a derived one using the rules for communication, abstraction and restriction.

An advance over the CCS was the work of Milner, Parrow, and Walker on the π-calculus [63], in the interest of studying *mobility* for concurrent processes. Due to mobility, the underlying topologies of systems become malleable, and hence a more sophisticated means of manipulating links became necessary, resulting in the central notion of a *name*.

Names take the form of both *communication channels* and *variables*. Therefore, it is possible to pass names on channels and, along with the rule for replication, the π-calculus becomes computationally complete (processes can be defined recursively) and thus a universal model of computation.

The π-calculus and its extensions are influenced primarily by the paradigm of parallel execution via interleaving and sharing of resources: processes compete for shared resources and "take turns" in order to execute. An argument against this paradigm is that it deviates from the notion of *true* concurrency, i.e. the global parallel evolution of a system of processes.

One approach towards true concurrency came in the form of the *chemical metaphor* for concurrency, pioneered by Banâtre, Coutant, and Le Métayer with their work on Γ [11], a "language-like" semantics of concurrency. Within the chemical metaphor systems of processes are solutions of molecules which

are stirred and interact when they meet other compatible molecules.

This model of concurrency allows truly parallel execution because reactions are strictly local, as opposed to competing for a central execution location: any number of interactions can be taking place concurrently provided each interaction is among disjoint sets of molecules. Furthermore, the model is scalable, because adding disjoint sets of molecules does not affect the existing solution.

In order to exploit the chemical metaphor it was necessary to obtain a semantics for concurrent programs. This was achieved by Berry and Boudol with their work on the Chemical Abstract Machine or CHAM [17], which is, essentially, an algebraic interpretation of chemical semantics.

The authors noticed that the molecular solutions were multi-sets and the stirring mechanism provided the commutativity and associativity of parallel composition. Along with the rules for reaction of molecules, program execution can be represented as a multi-set term re-writing system, and hence operational semantics can be attributed.

However, the CHAM is not very expressive for two reasons: first, molecules must travel and mix and match indeterminably within the solution, thus making the stirring mechanism impossible to implement efficiently; second, new reaction rules cannot be added into the solution, and hence processes cannot be defined recursively.

The work of Fournet and Gonthier on the Reflexive Chemical Abstract Machine, or RCHAM [28], addressed these two issues. By imposing strict *locality* on reaction sites for molecules, enforced syntactically, it is possible to declaratively define destinations for molecules which travel there directly; consequently, reasonably efficient implementations of the stirring mechanism can be realised.

Furthermore, by including reflection, in the sense that molecules can introduce new reaction rules (higher-level molecules), the model becomes fully

computational and can serve as a universal model of computation, much like the π-calculus.

The RCHAM can also be described by the join calculus [29], a basis for language design. The join calculus is characterised by purely local synchronisation via linear pattern-matching, strong adherence to syntactic rules, and a bias towards practical implementation via a direct embedding of a mini ML-like functional language within its core.

Specifically, the join calculus uses ML's function binding and pattern-matching in order to efficiently implement synchronisation in a declarative manner. Hence, messages (method invocations) travel directly to their destinations (are dispatched to their receivers), where they can interact after the synchronisation constraints of a join pattern have been satisfied.

The equational theory of the join calculus is based primarily on results of concurrency theory, which provides a systematic, strong means of stating and proving properties of concurrent programs. Although based on the π-calculus, it is not a direct extension of the calculus, because the CCS-based communication semantics of the latter are not compatible with the de-coupling of transmission and synchronisation of messages which lie at the core of the chemical metaphor.

2.1.2 Programming Concepts and Languages

Object-oriented features can be reasonably encoded in the join calculus via record-like structures, and properties common to objects can be effectively modelled, such as initialisation (constructors), self-reference (pointer to *this*), and specialisation (inheritance).

The JoCaml [27] language is a direct implementation of the join calculus which features join semantics and objects in the functional setting of OCaml. However, JoCaml constructs extend those of OCaml, as opposed to being compiled into OCaml libraries, and hence JoCaml is an extension to OCaml.

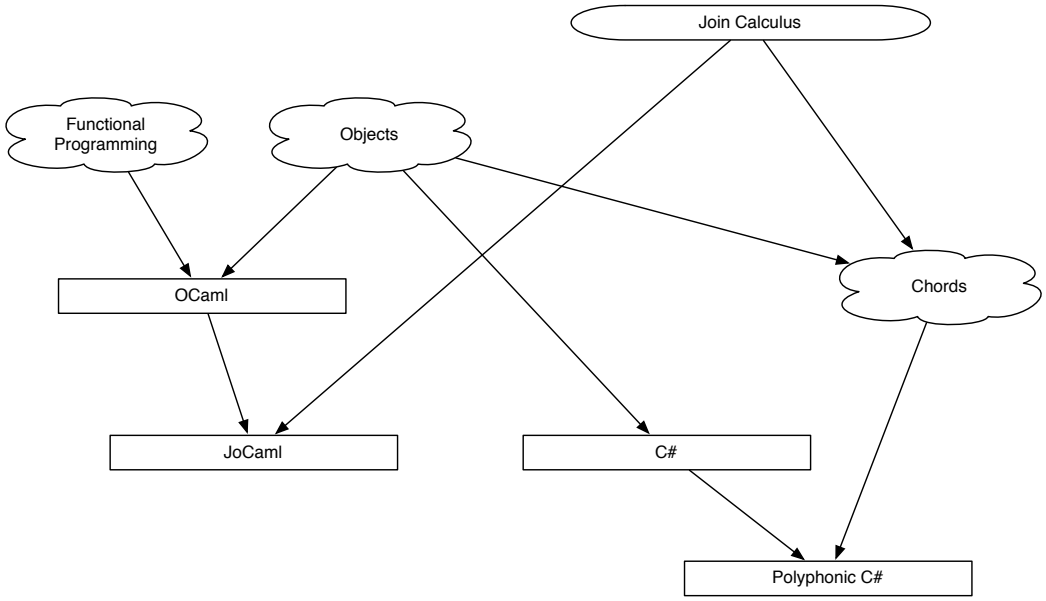

Figure 2.2: Programming Concepts and Languages Leading to Chords and Chorded Languages

The implementation of JoCaml is reasonably efficient (exploiting the existing native byte-code implementations of OCaml) and demonstrates how functional, imperative, object-oriented and join notions can be combined in a language which is based on concurrency, mobility and distribution.

An interesting experiment consisted of the work of Benton, Cardelli, and Fournet on introducing join semantics into the modern object-oriented language C$^\sharp$, resulting in Polyphonic C$^\sharp$ [15, 16] (and later part of C$_\omega$ [14]).

The authors point out that the chemical metaphor for concurrency and the object metaphor for programming are compatible conceptually and in terms of programming languages based on the combination of concepts from both metaphors.

Generally, the molecular representation of processes can be extended to objects, and hence objects become first-class components. Specifically, meth-

ods become communication channels, method invocation becomes message-passing, and objects provide higher-order names and reflection: an object can be sent to another object (similar to a name being sent to another name), and object specialisation can introduce new program behaviours (similar to new reaction rules).

The notion of join manifests in terms of *chords*, which are, essentially, collections of methods whose simultaneous invocations result in further program execution. Chords consist of a header which specifies the methods necessary for further execution, and a body of instructions which is to be executed.

Usually methods are invoked *asynchronously*, akin to messages being sent into the chemical solution. The detection of method invocations is the pattern-matching of appropriate messages, and execution of a chord is join-pattern satisfaction. Program execution generally blocks until a chord joins, thus modelling synchronous behaviours as well.

The reason chords are an interesting projection of join semantics is that they are introduced in a programming language setting which is fundamentally imperative in nature. The specification of methods and their combinations as chords are declarative and remain static within class definitions, but their runtime behaviour may depend upon such arbitrary constructs as monitors and explicitly controlled thread life-cycles.

2.2 Join Semantics and Chords

We now explore in greater detail the literature related to the calculi and languages which lead to the development of chords and chorded languages. We will use these results as the basis upon which we derive a formalism for chords in the remainder of this thesis.

We begin with the Join Calculus, a description of the RCHAM suitable for implementation as a programming language. The main implementation

is JoCaml [27], a functional language which features join semantics, imperative constructs, objects, failure detection and handling, and mobility. We will mention particular issues related to JoCaml, namely, implementation considerations and inheritance semantics. Further theoretical work involves object-orientation, resulting in the Objective Join Calculus. Furthermore, we will briefly look at functional nets and the language Funnel of Oderksy [69], which is similar to JoCaml.

We then focus on an informal implementation of chords on top of the modern object-oriented language C♯, resulting in Polyphonic C♯ (and later part of C_ω); we explore some common concurrency problems solved through chords, and consider implementation issues and important differences from the Join Calculus.

Finally, we give an overview of various experimental library-based implementations of chords in .NET, Scala and Java.

2.2.1 The Join Calculus

The Join Calculus [30, 29] is a language that models distributed and mobile programming. It was designed as a description of the RCHAM more suitable for implementation as a programming language, and is heavily influenced by the ML language and the π-Calculus.

Syntax and Semantics

The syntax of the core Join Calculus consists of processes, join definitions, and join patterns (see figure 2.3). The only expressions are variables; additional syntax for other kinds of expressions can be added to the core without affecting the essential properties of the calculus.

Messages consisting of zero or more names are sent as tuples to channels, and channels are names themselves. Join definitions are local to a process, and

P, Q, R	$::=$		processes
		$x\langle \tilde{u} \rangle$	asynchronous message
	$\mid$	def D in P	local definition
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	0	inert process
D	$::=$		definitions
		$J \triangleright P$	reaction rule
	$\mid$	$D \wedge D'$	composition
	$\mid$	T	void definition
J	$::=$		join patterns
		$x\langle \tilde{u} \rangle$	message pattern
	$\mid$	$J \mid J'$	synchronisation

Figure 2.3: The core Join Calculus syntax.

processes can be placed in parallel. A join definition consists of one or more join patterns in conjunction, and join patterns consist of one ore more message patterns in parallel.

A join pattern J is satisfied when there exists at least one message for each of the messages in J; therefore, if $J \equiv x\langle s \rangle \mid y\langle t \rangle$, the presence of an $x\langle u \rangle$ simultaneously with the presence of a $y\langle v \rangle$ satisfy J. A join definition D is satisfied whenever at least one of the join patterns it contains is satisfied, so if $D \equiv J \wedge J'$ and J was satisfied as above, D is satisfied. Whenever a join definition is satisfied the right-hand side of the definition replaces the messages which resulted in that satisfaction, so if for instance we had the definition def $J \triangleright Q \wedge J' \triangleright Q'$ in P and $P \equiv \ldots \mid x\langle u \rangle \mid y\langle v \rangle$ then P would evaluate to $\ldots \mid Q$, where the values u and v would appropriately substitute all instances of the variables s and t respectively.

The scope of s, t is Q, while the scope of x, y extends to the whole definition. We employ two auxiliary function $dv(D)$ and $fv(D)$ which give the defined and the free variables of D respectively. The structural congruence rlation $\equiv$ is the smallest relation such that for all processes P, Q, R, S, all definitions D, D',

and join patterns J such that $dv(D), dv(D')$ contain only fresh names (names are fresh with regards to a process when they do not appear free in the process [28]), we obtain:

$$\begin{aligned} P \mid Q &\equiv Q \mid P \\ (P \mid Q) \mid R &\equiv P \mid (Q \mid R) \\ P \mid \text{def } D \text{ in } Q &\equiv \text{def } D \text{ in } P \mid Q \\ \text{def } D \text{ in def } D' \text{ in } P &\equiv \text{def } D' \text{ in def } D \text{ in } P \end{aligned}$$

Furthermore, using an appropriate notion of α-equivallence $\equiv_\alpha$, in which non-free variables appearing in a process are implicitly renamed in order to avoid clashes with defined variables from join patterns during substitution [28], we obtain:

$$\begin{aligned} P \equiv_\alpha Q &\implies P \equiv Q \\ P \equiv Q &\implies P \mid R \equiv Q \mid R \\ R \equiv S,\ P \equiv Q &\implies \text{def } J \rhd R \text{ in } P \equiv \text{def } J \rhd S \text{ in } Q \end{aligned}$$

The reduction relation is defined as the τ-transitions of a labelled transition system $\xrightarrow{\delta}$ where δ ranges over $\{D\} \cup \{\tau\}$. The transition relation is the smallest relation such that for every definition $D = x\langle u\rangle \mid y\langle v\rangle \rhd R$:

$$x\langle s\rangle \mid y\langle t\rangle \xrightarrow{D} R\{{}^{s}/_{u},\ {}^{t}/_{v}\}$$

and for every transition $P \xrightarrow{\delta} P'$:

$$\begin{aligned} P \mid Q &\xrightarrow{\delta} P' \mid Q && \\ \text{def } D \text{ in } P &\xrightarrow{\delta} \text{def } D \text{ in } P' && \text{if } fv(D) \cap dv(\delta) = \emptyset \\ \text{def } \delta \text{ in } P &\xrightarrow{\tau} \text{def } \delta \text{ in } P' && \text{if } \delta = D \\ Q &\xrightarrow{\delta} Q' && \text{if } P \equiv P' \text{ and } Q \equiv Q' \end{aligned}$$

Thus it is possible to use the δ relation to define any operation found in other calculi by means of reactions.

Programming

Following are some examples of programs written in the syntax of the calculus; their aim is to serve as indicators that typical constructs one expects from a concurrent programming language are easy and intuitive to define in the join calculus.

Channel forwarding:

$$\text{def } x\langle u\rangle \ \triangleright \ y\langle u\rangle \text{ in } P$$

where message u sent on x is forwarded to y.

Channel multiplexing:

$$\text{def } x_1\langle u\rangle \mid x_2\langle v\rangle \ \triangleright \ y\langle u,\ v\rangle \text{ in } P$$

where messages u and v sent on x_1 and x_2 respectively are multiplexed and sent on y.

Higher-level channels:

$$\text{def } x\langle u\rangle \mid y\langle k\rangle \ \triangleright \ k\langle u\rangle \text{ in } P$$

where message u sent on x is forwarded to name k, obtained from y.

Non-deterministic choice:

$$\text{def } s\langle\,\rangle \ \triangleright \ P \wedge s\langle\,\rangle \ \triangleright \ Q \text{ in } s\langle\,\rangle$$

where internal non-deterministic choice of the form $P + Q$ is expressed with a compound definition.

Looping:

$$\text{def } loop\langle\,\rangle \ \triangleright \ P \mid loop\langle\,\rangle \text{ in } loop\langle\,\rangle \mid Q$$

where process P is replicated repeatedly.

And record-like encoding of a cell:

$$\begin{array}{rcl} & \text{def} & get\,\langle k\rangle \mid s\langle u\rangle \;\triangleright\; k\langle u\rangle \mid s\langle u\rangle \\ \text{def}\; mkcell\langle u_0,\; k_0\rangle \;\triangleright\; (& \wedge & set\langle u,\; k\rangle \mid s\langle u\rangle \;\triangleright\; k\langle\,\rangle \mid s\langle u\rangle \;) \\ & \text{in} & s\langle u_0\rangle \mid k_0\langle get,\; set\rangle \end{array}$$

where the names *get*, *set* are returned for use by the creator of the cell and s contains the internal state of the cell, enforced by the strict lexical scoping.

Inheritance

In [32, 33] Fournet et al. propose the Objective Join-Calculus, an extension of the Join-Calculus which addresses issues of inheritance and class type definitions[1]. Their approach consists of defining objects as collections of records, and imposing a new class layer on top of objects which enables incremental programming and code reuse.

In the Objective Join-Calculus method calls, locks, and states are treated in a uniform way using only asynchronous messages; classes are partial message definitions which can be combined and refined with a set of operators for behavioural and synchronisation inheritance. Specifically, a class can directly inherit all the behaviours of another class via the new syntax "class $c\;=\;C$ in P" where C is a class definition, and all refinements are performed through a set of refinement clauses S with the new syntax "match C with S end".

The following is an example from [33] of a one-place buffer:

$$\begin{array}{l} class\ \text{buffer} = self(z) \\ \qquad \text{get}(r)\ \&\ \text{Some}(n) \;\triangleright\; r.\text{reply}(n)\ \&\ z.\text{Empty}() \\ \qquad or\ \text{put}(n, r)\ \&\ \text{Empty}() \;\triangleright\; r.\text{reply}()\ \&\ z.\text{Some}(n) \end{array}$$

where the new syntax for classes contains the keywords *class* and *self* which are used to explicitly bind the name z to $self$; then, buffer objects can be instantiated as follows:

[1] In this calculus the symbol & is used instead of | in join patterns and the symbol *or* instead of $\wedge$ in join definitions

$$obj\ b\ =\ \text{buffer}\ init\ b.\text{Empty}()$$

where the keywords *obj* and *init* define a new object and its initialisation logic respectively. The buffer is now extended to include logging, which demonstrates a disjunction of an inherited class and new reactions:

$$\begin{aligned}
&class\ \text{logging_buffer} = self(z)\\
&\qquad \text{buffer}\\
&\qquad or\ \text{log}() \ \&\ \text{Some}(n)\ \triangleright\ print(n)\ \&\ z.\text{Some}(n)\\
&\qquad or\ \text{log}() \ \&\ \text{Empty}()\ \triangleright\ print(\text{“}Empty\text{”})\ \&\ z.\text{Empty}()
\end{aligned}$$

where the existing behaviours from the *buffer* class compete with the new behaviours defined in the *logging_buffer* class. Finally, an example of the *match* operator, which finds all instances of a term and replaces them with new terms, such as printing all placement of elements in the buffer:

$$\begin{aligned}
&class\ \text{all_logging_buffer} =\\
&\qquad match\ \text{buffer}\ with\\
&\qquad\qquad \text{put}\ (n,r)\ \Longrightarrow\ \text{put}\ (n,r)\ \triangleright\ \text{print}\ (n)\\
&\qquad end
\end{aligned}$$

demonstrating the new keywords *match*, *with* and *end*, and the operator $\Longrightarrow$; the resulting class *all_logging_buffer* is the same as *logging_buffer* where all instances of put (n,r) have been replaced with put $(n,r)\ \triangleright\ \text{print}\ (n)$.

The work of Qin et al. [51, 52, 53, 50] has further elaborated and simplified the semantics of inheritance in the Join-Calculus, with the aim of reaching a reasonable implementation of the Objective Join-Calculus. The authors point out several shortcomings of the previous work, namely, that objects are not polymorphic enough, classes contain some synchronisation information which is not as expressive as necessary to encode some behaviours, some abstract classes may in fact contain concrete definitions, and under some conditions, extremely complicated runtime checks are necessary in order to ensure program safety. The changes proposed consist of a new type system in which all

synchronisation information is included within class types, and a new algebraic pattern matching system.

2.2.2 JoCaml

The JoCaml language [27], originally implemented by Fessant et al. [25] and then re-implemented by Fournet et al. [29], is an implementation of join semantics based on the OCaml language. The examples below are adapted from the language manual, available from [29].

JoCaml features processes and expressions, the former being executed asynchronously and producing no result value, while the latter being evaluated synchronously and resulting in values. Processes communicate by sending values onto channels, which are themselves values. Local synchronisation is made possible through OCaml's function binding and pattern matching. Typing is very similar to that of ML [31].

The following is an example of a JoCaml program:

```
1 def fruit(f) & cake(c) = print(f ^ " " ^ c);
```

where a single join pattern is defined requiring the simultaneous presence of values for the channels *fruit* and *cake*, denoted by the ampersand sign. By invoking these two methods, as in the following example:

```
1 spawn fruit("apple") & cake("pie");
```

the string *apple pie* is printed. The keyword *spawn* creates new processes which are executed asynchronously; hence, the ampersand here separates independent processes rather than names participating in a join definition.

The following is an example of choice between multiple join patterns:

```
1 def apple() & pie() = print("apple pie");
2  or raspberry() & pie() = print("raspberry pie");
```

If there are invocations for both *apple* and *raspberry*, either of the strings *apple pie* or *raspberry pie* may be printed:

```
spawn apple() & raspberry() & pie();
```

Synchronous channels can be encoded by means of continuations, as in the following example:

```
def succ(x, k) = print(x); k(x+1);
```

where the join pattern prints the value of x and then "passes it on" to a continuation k after incrementing its value by one. However, for convenience, JoCaml also features synchronous channels by means of an explicit *return* operation, as in:

```
def succ(x) = print(x); reply x+1 to succ
```

Pattern matching enables further discrimination between multiple join patterns depending on the type of argument a channel expects, as in the following example:

```
type fruit = Apple | Raspberry | Cheese
type desert = Pie | Cake
def f(Apple) & d(Pie) = print("apple pie");
 or f(Raspberry) & d(Pie) = print("raspberry pie");
 or f(Raspberry) & d(Cake) = print("raspberry cake");
 or f(Cheese) & d(Cake) = print("cheese cake");
```

where the two join patterns `f(Raspberry)&d(Pie)` and `f(Raspberry)&d(Cake)` are distinct.

Implementation

In [55] Maranget et al. observe that join patterns are intended to be heavily used by programmers of languages such as JoCaml; this implies that the compilation and runtime management of join pattern satisfaction become an important factor in the implementation of a join-based programming language.

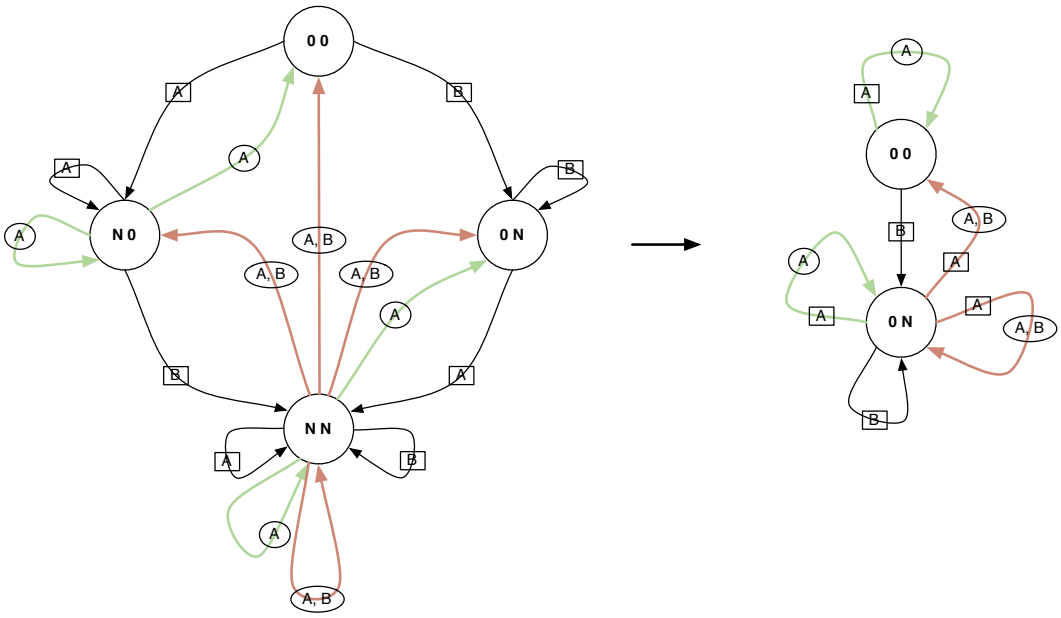

Figure 2.4: A non-deterministic join automaton and its reduction.

Their approach consists of transforming join definitions into *join automata*, where states represent a *matching status* for a join definition and transitions represent the issuance of names (or method calls). The matching status consists of counters for each of the participating names in the join definition, and generally takes values of either 0 or N signifying, respectively, no availability and at least one issued name (invoked method). Once a state where each name has the value of N is reached, the system can choose to perform a join.

A transition is followed whenever a name is issued (hence the automata are complete, in the sense that a transition for each of the participating names must be present). Furthermore, each time a matching occurs and the join pattern is satisfied, a "decreasing" transition is followed which places the system in a state where possibly some names are no longer available (if the last name was consumed for the join), represented by the value 0.

For instance, the following JoCaml program:

```
let A() = print("A");
let A() | B() = print("AB");
```

can be compiled into the non-deterministic join automaton in the left of figure 2.4, where the two counters in each state correspond to the names A and B respectively, square transitions represent the invocation of names, and oval transitions represent the consumption of names due to joining. The left and bottom states correspond to possible joins of the first and second patterns, respectively; the bottom state presents a non-deterministic choice between either pattern, as both can join.

The authors observe that automata can become quite large and result in high overheads; therefore, they propose two changes: *eager join* semantics and *loss of behaviours*. Since the choice between competing join definitions is left unspecified in the Join Calculus, an implementation of such semantics can legally reduce or even eliminate choice. Hence, join automata can be reduced by always letting the first join pattern capable of joining to proceed; for instance, in figure 2.4 the right automaton is such a reduction of the left, where the first join pattern "eagerly" joins whenver an invocation of A becomes available.

The bottom state, however, still exhibits non-deterministic choice between the two patterns; this can further be eliminated by requiring, for instance, the second pattern to always join while the system in this state (there are B messages available). Although this is a legal implementation of the Join Calculus, it inevitably leads to loss of behaviours, as in the case of the following JoCaml program:

```
let A() = print("A");
let B() = print("B");
let A() | B() = print("AB");
```

where the third pattern will never join under eager semantics. Whether this

```
def startRead  & noWriters =  startRead1,
    startRead1 & noReaders =  () & noWriters & readers 1,
    startRead1 & readers n =  () & noWriters & readers (n+1),
    endRead    & readers n =  if (n == 1) then noReaders
                              else (readers (n-1)),
    startWrite & noWriters =  startWrite1 & writers 1,
    startWrite & writers n =  startWrite1 & writers (n+1),
    startWrite1 & noReaders = (),
    endWrite   & writers n =  noReaders &
            (if (n == 1) then noWriters
      else (writers (n-1)));
noWriters & noReaders
```

Listing 2.1: Example of a readers / writers program in Funnel.

is a reasonable implementation of JoCaml depends on the way programmers intend to use their join definitions, but is not further discussed in this work.

2.2.3 Functional Nets and Funnel

In [67, 68, 69] Odersky et al. observe that no contemporary model of computation addresses concurrency in a satisfactory manner; furthermore they notice that object-oriented paradigms are suitable for structuring large programs so that they may facilitate reuse (for instance, encapsulation, inheritance, subtyping, and so on). They propose a model of computation called functional nets, which has resulted in the language Funnel; they claim this model is simple, intuitive, and combines well with object-oriented paradigms.

Functional nets combine imperative, functional, and concurrent programming constructs, and are based on the pattern-matching semantics of join: functions are composed in parallel, and calling a function blocks until each of the other related functions is enabled; the right-hand side of a function (the body of code) then replaces the left-hand side (the declaration) where param-

eters have been appropriately mapped. For instance, a one-place buffer looks like this:

```
def put x & empty =  () & full x
   get  & full x =  x  & empty
```

Functional programming becomes a special case of pattern-matching where no parallel definition constrains the execution of a function, such as the following implementation of a map:

```
def map (f, xs) =
    if isEmpty xs then Nil
    else Cons (f (head xs), map (f, tail xs))
```

Imperative programming is achieved by creating mutable variables much like the cells of the Join Calculus, where an internal state is maintained:

```
def newref init = {
   def read   & state x  =  x  & state x,
       write y & state x =  () & state y;
   (read, write) & state init
}
```

Using such variables systematically, an object with methods $m_1, \ldots, m_n$ and fields $x_1, \ldots, x_n$ can be encoded as such:

```
def m1 & state (x1,...,xn) = ... ; state (y1,...,yn),
                             ...
   mn  & state (x1,...,xn) = ... ; state (z1,...,zn);
(m1,...,mn) & state (init1,..., initn)
```

where $(m_1, \ldots, m_n)$ is the result and $(init_1, \ldots, init_n)$ some initial state; this encoding enforces mutual exclusion in state updates. As a concrete example, consider a readers/writers solution in Funnel from listing 2.1.

2.2.4 The Language Polyphonic C♯

In [15, 16] Benton, Cardelli, and Fournet present the prototypical programming language Polyphonic C♯, an extension of C♯ with two new constructs: *asynchronous methods* and *chords*, or join patterns on sequences of method invocations.

Rather than extending the objective join calculus into an object-oriented programming language, the authors extended a generic imperative language with certain features of the join calculus. Asynchronous methods are similar to named channels which receive asynchronous messages, and chords are a constrained version of join definitions.

Asynchronous Methods

In generic imperative object-oriented languages, methods are synchronous: an invoking thread will hand over execution control to the body of the method it invokes, and only receive back control when the method body finishes execution. After invocation returns, the caller is left with the return value of the method (or *void*).

Asynchronous methods, on the other hand, return immediately to their calling thread. The invocation itself is indistinguishable from an invocation of a synchronous method: a message containing the actuals corresponding to the method's formals is created and despatched, and the method immediately leaves the caller with void as the return value.

From a memory model point of view, invoking an asynchronous method means that somehow the message comprising the invocation must be issued, despatched, and queued at the receiving end. This results in a queueing system which collects asynchronous method invocations and places their messages in queues associated with the methods at the receiving object. Hence, in terms of memory shape, objects contain fields and queues of messages.

From a type system point of view, a new return type for asynchronous methods is introduced: *async*. In order to preserve soundness, the new type is a subtype of *void*. This, as we will see later on, leads to interesting issues when combined with overloading and inheritance.

An asynchronous method generally looks like this:

$$async \text{ methodName } (\ \ldots\ arguments\ \ldots\ )\ \{\ \ \ldots\ body\ \ldots\ \ \}$$

The body of the method must then be scheduled for execution. Obviously, there is no thread to synchronously wait for the complection of execution as the invoking thread has continued immediately after the invocation. Hence, the body of the asynchronous method executes in a new thread.

Chords

Chords are similar to join definitions: a chord consists of a header and a body. The header describes a pattern of methods whose joint invocation (the simultaneous presence of invocations) functions as a guard on the execution of the body. The body consists of a set of sequential instructions.

An example of a chord is the unbounded buffer:

```
1 class UnboundedBuffer {
2   Object get() & async put(Object o) { return o; }
3 }
```

Invocations to put(*Object o*) are asynchronously queued at the receiving UnboundedBuffer object. Whenever a thread invokes get( ) the chord must join before the body is executed. As long as there are message in the queue of "put" then joins can occur and the body can be scheduled for exeuction.

If there are no messages in the queue, then the call to "get" will block until a join is possible. This is a case of a synchronous chord, or a chord which contains a sycnhronous method in its header. Another kind of chord is an

asynchronous one, or one which contains only methods of type *async* in its header:

```
1 class Adder {
2   async adda(int a) & async addb(int b) { print(a + b); }
3 }
```

This chord's body will, again, execute when there is a simultaneous presence of messages on both "adda" and "addb" queues. However, in this sitution there is no thread blocking on a call to one of the methods. Hence, it is up to the scheduler to spawn a new thread for the body to execute (or use a free thread from a pool).

Chords may have at most one synchronous method in their headers, as this simplifies scheduling semantics. Furthermore, headers must be linear, so an asynchronous method may appear at most once in any header (similar to the join calculus restriction of linearity in names appearing in join patterns).

Non-Determinism

Following the notion of multisets as encodings of chemical solutions in the join calculus, asynchronous method queues behave non-deterministically. For instance, in the following example sequence of invocations:

```
1 UnboundedBuffer buffer = new UnboundedBuffer();
2 buffer.put("Hello");
3 buffer.put("World");
4 print(buffer.get() + " " + buffer.get());
```

both "Hello World" and "World Hello" are valid possible program behaviours.

Furthermore, a second class of non-determinism comes in the choice of message despatch, much like in the join calculus, when more than one chord headers contain a method. So, for instance, in the following example:

```
1 class ChoiceBuffer {
2   int get() & async put(int n) { return n; }
```

```
3   String get() & async put(int n) { return toString(n); }
4 }
```

there is a single queue for "put", however, there is now a choice between which of the two chords should join (assuming calls are pending on both of the "get" methods). This decision is made by the scheduler and remains abstract in the definition of chords.

Overloading and Inheritance

The introduction of chords gives rise to a new kind of overloading, that of join patterns. Consider the following two classes, the second of which overloads the single chord:

```
1 class Parent {
2   void f() & async a() { ... }
3 }
4 class Child extends Parent {
5   void f() & async a() & async b() { ... }
6 }
```

in this situation the scheduler must consider the runtime type of the object, since it is possible for both of the chords to join when there is a simultaneous presence of messages in the queues of the "a" and "b" methods. If two threads are blocked on an invocation of "f", and one thread is using the object with the "Parent" runtime type, and the other with the "Child" type, then there is a non-deterministic choice between the two joins.

There are another two kinds of overloading possible; consider the following two classes which extend "Parent":

```
1 class ChildA extends Parent {
2   void f() & async c() { ... }
3 }
4 class ChildB extends Parent {
5   void f() { ... }
```

```
6 }
```

In the case of "ChildB", the chord has retained only its synchronous method "f", so the choice is less complicated than the previous case (it is clear which chord *can* join). In the case of "ChildA", however, we have gone back to the complication of deciding whether we immediately join the chord which contains only "f" (and can trivially join) or waiting for a message to appear on the queue for "a".

Hence, we are now presented with two kinds of overloading. First, we can overload methods by altering their argument lists in unique ways. Second, we can overload chords by changing the join patterns of methods in unique (and strictly linear) ways.

The approach used by the authors in the handling of this increasingly complex overloading problem is to maintain chords local to class declarations. In this respect, any class which "partially" modifies a chord must also modify all of the affected chords of the superclass. For instance:

```
1 class Parent {
2   void f() & async a() { ... }
3   void f() & async b() { ... }
4 }
5 class Child extends Parent {
6   async a() { ... }
7 }
```

The class "Child" has overriden "a", but has also "half" overridden the method "f", since it appears in a chord with "a" in it. If this kind of inheritance were permitted then the inadvertent introduction of deadlock (or *async leakage*) would be an all too common problem.

Specifically, code written to expect instances of "Parent", which matches invocations of "f" with "a" in the join pattern would probably fail to work when passed an instance of "Child", as calls to "a" would immediately be

despatched to the asynchronous method on its own and calls to "f" would starve.

In Polyphonic C♯ the extending class must extend the first of the two chords, even if this means simply copying over the code. There are many more issues with inheritance, such as explicit tagging of *virtual*, *overides*, and *private*; see [16] for examples of these problems and the simplifications in the design of the language which were made in order to avoid them.

Language Characteristics

Assuming a typical grammar for a language such as C♯ (there isn't a formal definition published yet), the language is extended as follows:

$$
\begin{array}{rcl}
ReturnType & \stackrel{\text{def}}{=} & Type \mid void \mid async \\
ChordDeclaration & \stackrel{\text{def}}{=} & MethodHeader\ [\&\ MethodHeader]^*\ Body \\
MethodHeader & \stackrel{\text{def}}{=} & Attributes\ Modifiers\ ReturnType\ Name\ (Formals)
\end{array}
$$

Notice that the language contains methods and chords. It is unclear whether a chord's body should be considered as its synchronous method's body (and what happens if the chord's header does not contain a synchronous method?). It is also unclear whether a method on its own should be considered a degenerate case of chord with a single synchronous method as its join pattern. This vagueness stems from the fact that Polyphonic C♯ has not been formally specified.

An important issue to consider is well-formedness. Beyond the normal constraints of linearity in chord headers mentioned previously, it is fundamental that asynchronous method formals are never of type "ref" or "out". Neither of these kinds of parameters make sense for an asynchronous message, since asynchronous messages belong to an invocation which is handled locally by a new thread; the original thread which made the call may not exist any more and neither would its "ref" or "out" variables.

This is a more general problem with store-based imperative languages. Since messages are considered to be purely local, there should never be refer-

ences to an object in more than one thread; this can be recognised as a case of breaking the membrane law of the RCHAM: a molecule, or message, can be accessed from outside its membrane.

Furthermore, inheritance in Polyphonic C$^\sharp$ lead the authors to include covariant return types in the case of *async*. Specifically:

- An *async* method may override a *void* method.
- A *void* delegate may be created from an *async* method.
- An *async* method may implement a *void* method in an interface.

These properties are not very surprising, and according to the authors make the language more intuitive.

Important Differences From The Join Calculus

Polyphonic C$^\sharp$ is not a language with joins as the concurrency primitive. The chord definitions are, essentially, meta-data which are used by a scheduler programmed to emulate join patterns.

The language itself is pre-processed by a compiler which translates it into C$^\sharp$, and explicitly uses imperative constructs such as threads and monitors to emulate the appropriate scheduling (see the paper for the source code of the translation).

However, implementation issues aside, the language differs from the join calculus in concept at two primary levels: no higher-order definitions and no inherent parallelism in expression bodies.

Firstly, because the language is based on value passing, and not name passing, it is not possible to refer to chords. This means that join calculus operations such as channel forwarding (def $x\langle u\rangle \mid y\langle k\rangle \ \triangleright \ k\langle u\rangle$ in P) are not possible. An extension of the language with canonical names for chords would be necessary for this kind of higher-order constructs.

Secondly, there is no parallel compositional operator like "|". Instead, a body of code which requires to be split into parallel parts must be explicitely separated into two additional asynchronous methods which can be sequentially invoked.

In this respect, there is a significant difference between chords and the objective join calculus: where in the calculus a sequence of instructions was translated into an implicit continuation, in Polyphonic C♯ a parallel composition must be explicitely translated into forking continuations.

Implementation Issues

As we mention above, Polyphonic C♯ is translated into C♯. Hence, the management of threads and emulation of asynchronous messages, as well as the joining of invocations (the *rendezvous*) are implemented by segments of code generated and embedded along with the user's program.

This explicit management of scheduling with imperative constructs may lead to dangerous premature optimisations. The authors faced one such problem, which involves the way in which a join is detected.

An eager evaluation strategy for determining when a join is ready to happen would consist of generating automata which model the simultaneous existence of messages in the queues of asynchronous methods. These automata will "accept" when all the queues in a chord definition have at least one message.

However, if these automata perform a transition immediately after an invocation of an asynchronous method, then starvation may be inflicted on the program. Consider:

```
1 class SchedulerTrap {
2   void m1() & async s() & async t() { ... }
3   void m2() & async s() { ... }
4   void m3() & async t() { ... }
5 }
```

and say that four threads execute the following sequence:

Thread 1: call m1() and block.

Thread 2: call m2() and block.

Thread 0: call t() and then s(), waking thread 1.

Thread 3: call m3() and succeed, hence consuming t().

Thread 1: still blocked on m1().

Thread 3 preempts thread 1 and "steals" its message on "t". The remaining message on "s" suffices to join thread 2. However, if neither thread 3 nor thread 1 awake thread 2, then we have a race condition leading to deadlock.

The solution is to ensure that in situations such as thread 3, the queues of the asynchronous methods which appear in the chord's header are scanned again for messages. This implies that the automata should either not perform eager transitions or be ready to "backtrack" (check for input again).

2.2.5 Library Implementations of Join Concurrency

We briefly explore several library-based implementations of chord-like functionality in languages such as Visual Basic, Java and Scala; these approaches typically extend the normal language syntax with join-like syntax, implement a compile-time transformation, and utilise chord-like logic in ad-hoc libraries which are distributed with the resulting program.

The Joins Concurrency Library

In [78] Russo presents the *Joins Concurrency Library*, a library-based implementation of join concurrency for the .NET framework (currently compatible with C♯ and Visual Basic). Using generics, the library is made available for the entire family of .NET languages.

The Joins library is based on a central *Join Class*, which provides a declarative mechanism for defining thread-safe asynchronous and synchronous commu-

```
1  class Buffer {
2    public readonly Asynchronous.Channel<string> Put;
3    public readonly Synchronous<string>.Channel Get;
4    public Buffer() {
5      Join j = Join.Create();
6      j.Initialize(out Put);
7      j.Initialize(out Get);
8      j.When(Get).And(Put).Do(delegate(string s) {
9        return s;
10     });
11   }
12 }
```

Listing 2.2: Example of a buffer using the Joins Library.

nication *channels*. Channels are instances of a special delegate type, initialised from a common *join object*; communication patterns and their effects are declared using join patterns, which consist of bodies of code guarded by linear combinations of channels. Bodies of code (which can also be seen as continuations) are provided by the user as delegates which can manipulate resources protected by the object which contains them. Communication and synchronisation is achieved by invoking delegates (and passing arguments through the invocations), and optionally waiting for them to return a value.

Listing 2.2 is an example of a string buffer for shared communication between threads. There are two communication channels, the `Put` and `Get` fields, which contain an asynchronous and a synchronous channel, respectively. A producer thread asynchronously sends a string by invoking `Put(s)` without blocking; a consumer thread synchronously requests a string by calling `Get()`, with the possibility of blocking until a string is available. Strings are passed between threads atomically.

```
class SyncChan[a] with {
  join {
    def put(x: a): Unit;
    def get: a;
    put(x) & get = spawn < (reply to put) | (reply (x) to get) >;
  }
}
```

Listing 2.3: Example of a synchronous channel using the Scala join compiler.

Join in Scala

Scala has rudimentary support for chords through an extended syntax and associated compile-time transformations implemented as an additional compilation phase within the standard compiler *socos*. The chord-like functionality is in an experimental state and typically supports synchronisation rules through pattern-matching of functions (or methods) [22].

Listing 2.3 consists of a synchronous channel in Scala: the new keyword *join* enables the definition of several methods and patterns which execute as continuations when the appropriate method calls are detected.

Other Implementations

In [19] Chrysanthakopoulos et al. present an asynchronous messaging library for C♯ which can be used to express complex *join* patterns without the need for language extensions. The library allows hierarchical composition of such patterns, and many common synchronisation patterns are demonstrated, such as multiple readers and writers; their claim is that using this library will enable future investigation of concurrency idioms without having to resort to language-level changes for the accommodation of chords.

There are two transformation-based approaches to introducing chords into Java by Wood [91] and von Itzsein [89, 88, 90], the latter called *JoinJava*.

Both approaches transform augmented Java source-code into normal source-code which includes chord-like logic and calls to ad-hoc libraries.

2.3 Choice, Scheduling, and Fairness

A realistic implementation of a concurrent language will treat individual processes in a consistent manner, and will manage resources (such as allocation of processor time or delivery of messages) in a predictable way. 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. In this section we give an overview of the problems arising from non-deterministic choice in programming language constructs, and the kinds of properties schedulers handling such choice may be required to guarantee.

2.3.1 Reactive Systems, States and Behaviours

Due to their complexity, concurrent systems do not lend themselves to description as a straightforward function of input to output; rather, such systems are described in terms of their permissible behaviours [2]. Behaviours are represented as sequences of states, and a system changes state when one of its components performs a computation or when a component responds to an event issued by other componentns or the system's environment. Hence, we are interested in the specification of behaviours for *reactive* systems.

2.3.2 Transition Systems and Execution Sequences

This method of specification naturally leads to the use of state machines, or transition systems, where transitions between states represent computations or reactions of the system to events. A particular sequence of states is called an *execution* of the system (or alternatively a *run*). Executions can be *finite* or *infinite* sequences, usually over a fixed countable set of *states*, Σ. Σ^+ denotes the set of finite executions, Σ^ω denotes the set of infinite executions, and $\Sigma^+ \cup \Sigma^\omega = \Sigma^\infty$ denotes the set of all executions. We conventionally use $\alpha, \beta, \ldots$ to denote finite executions, and $x, y, \ldots$ for arbitrary ones. Concatenation of executions is denoted by juxtaposition. Any set of sequences is a *property* of the transition system, and we write $x \vdash P$ to denote that an execution sequence x satisfies (or is contained in) the property P.

2.3.3 Safety and Liveness Properties

A *safety property* of a transition system establishes that "something bad will not happen" during execution. A *liveness property* of a transition system establishes that eventually "something good must happen" during execution. The distinction between *safety* and *liveness* properties was first proposed by [46] and later formalized by [47] and [4]. The notions are based on [82], and have become well established in the specification, analysis and verification of reactive systems [75]. Examples of safety properties include: partial correctness, mutual exclusion, deadlock freedom, and first-come-first-serve; examples of liveness properties include: starvation freedom, termination, and guaranteed service.

Safety Properties

In [47] a property is termed a safety property if and only if each execution violating the property has a finite prefix which also violates it, thus expressing

the intuition that the "bad thing" can be detected in a finite initial sequence of the execution, and the occurrence of a "bad thing" in a prefix of the execution is *irremediable*. The converse also holds: if a finite prefix of an execution violates the property then so does the execution. However, [4] represent a finite prefix of an execution as the set of all continuations from that point onwards, leading to a more general notion of safety properties; hence, the following definition of a safety property:

$$\forall\ \alpha \in \Sigma^{\omega}\ :\ \alpha \not\models P\ \Longrightarrow\ \exists\ i, 0 \leqslant i\ :\ \forall\ \beta \in \Sigma^{\omega}\ :\ \alpha_i\beta \not\models P$$

The above definition does not specify what the "bad thing" is, other than requiring it to be *discrete*: there is an identifiable point in the sequence of the execution at which it happens. Also, the definition can only stipulate that something happens always (as opposed to sometime), in this case the property ¬"bad thing".

[42] observes that a safety property can be *generated* by a transition system with finite internal non-determinism [2], in the sense that the transition system specifies all the permissible behaviours of a system, i.e. what can and cannot happen during execution. Therefore, a safety property is obtained through the combination of the initial conditions of a program and the next-state relation.

Liveness Properties

A property is termed a liveness property if and only if it contains at least one continuation for every finite prefix, corresponding to the intuition that the "good thing" can still occur after any finite execution, called *responsiveness* by [54]. Indeed, if a partial execution were irremediable, then it would be a "bad thing". Hence, the following definition of a liveness property:

$$\forall\ \alpha \in \Sigma^{+}\ :\ \exists\ \beta \in \Sigma^{\omega}\ :\ \alpha\beta \vdash P$$

The above definition, similarly to the definition of safety, does not specify what a "good thing" is; furthermore, it does not require the good thing to be discrete either: the liveness property can be a collection of infinite discrete events, such as in the case of *progress*.

Decomposition

Safety properties are closed under conjunction and finite disjunction, and hence a characterisation of safety and liveness properties corresponds to a topology where safety properties are the closed sets and liveness properties are the dense sets. Therefore, we can express each property $P = \langle S, L \rangle$ where S is a safety property and L is a liveness property. This result was used by [4] in order to show that every property can be expressed as a conjunction of a safety and a liveness property, resulting in the *decomposition theorem*. A proof of the decomposition theorem without use of the topological characterisation was given by [80].

2.3.4 Interleaving Concurrency, Choice and Scheduling

By definition, concurrent systems will have several components ready to execute at any given time (and hence several transitions will be eligible). Hence, the notion of *choice* arises when determining which component will execute next, and this further leads to *interleaving* of executions: each component executes its constituent atomic operations in order, however, the relative order of execution among the components of the system is unspecified.

Moreover, although it is desirable that models of concurrent systems imply nothing of the relative execution speeds of their components, complications arise when a component cannot proceed autonomously: other components may prevent it from executing or it may need to interact with another component first. This problem of choosing which component executes next (or

which transition occurs next), relevant to models of concurrency which feature non-deterministic interleaving of computation, can be resolved by an arbiter external to the system, or *scheduler*. [2]

2.3.5 Abstraction Through Non-Deterministic Choice

It is possible to abstract from relative execution times and particular scheduling policies by introducing non-deterministic choice. Hence, concurrent programs are reduced to a single executing component with non-deterministically chosen continuations. What constitutes a continuation depends on the kinds of transition available in the system. Furthermore, non-determinism in reactive systems can also be used to abstract from the interaction with the environment. Indeed, many authors study properties of concurrent systems only in the form of non-deterministic choice [6].

If, however, we use non-determinism to abstract too much we may end up with systems which allow behaviours that no implementation of a scheduler would allow, nor any realistic environment would exhibit. For instance, pathological execution patterns can be considered which, although permissible, are unlikely to manifest under a reasonable scheduler implementation; e.g. a particular component of the system never executes despite being eligible to do so throughout a long enough execution (the precise definition of what constitutes a reasonable scheduler and under which circumstances an execution is considered long enough will be given shortly).

2.3.6 Fairness Assumption

Therefore, it is desirable for a system to imply that each of its components always eventually proceeds unless it has terminated. Guaranteeing that anything

[2]Non-interleaving models of concurrency, or *cooperative concurrency*, is beyond the scope of this work; for a survey of properties of non-interleaving models of concurrency see [44].

at all will happen, or that particular choices will eventually be made, requires a liveness property, usually called a *fairness assumption* of the system [70, 64]. In terms of choice, the fairness assumption relates to the non-deterministic satisfaction of guards, and is called *unbounded non-determinism* in [23].

An intuitive definition of fairness, expressed in terms of a system of concurrent components, is that no component which is capable of proceeding sufficiently often should be delayed indefinitely. A specific fairness property of a system will depend on the interpretation of "capable of evaluating" and "sufficiently often". The former determines which part of a system can evaluate, and is called the *granularity level* of fairness; the latter determines the conditions under which a component will proceed, and is called the *strength* of fairness [45]. In terms of choice, fairness means a particular choice is made sufficiently often provided that it is sufficiently often possible [5]. Therefore, a reasonable scheduler is one which allows only those executions which are considered "fair", and the notion of "long enough" is a direct consequence of the interpretation of "sufficiently often".

2.3.7 Defining Fairness

The above intuitive definition of fairness, however, lacks precision: what is the most general sense of "sufficiently often"? What does one mean by "possible"? Can we consider a notion more general than transition? Any definition of fairness must address these issues [87, 85]. It is thus desirable to obtain the most general definition of fairness possible, which is independent of the semantics of the underlying transition system.

Criteria for Fairness

As a first step towards such a definition, [5] give three criteria which must be satisfied by every notion of fairness: *feasibility*, *equivalence robustness*, and

liveness enhancement.

Any notion of fairness will exclude executions deemed "unfair" which otherwise would constitute legal executions of the underlying transition system; feasibility requires that there always be some legal executions remaining after such exclusion, for every legal program. Indeed, any implementation of a scheduler could not correctly treat a notion of fairness without the feasibility requirement, as it would not be possible to predict all the eligible continuations of a program at each step of an execution, and thus it would be possible for the scheduler to "paint itself into a corner"; therefore, it must be possible to extend each partial execution to a fair one.

In the presence of non-deterministic interleaving, the execution order of independent computation steps is arbitrary, and it is thus desirable for any two linear execution sequences which are identical up to the order of two independent computations to be equivalent. Equivalence robustness requires that any notion of fairness respects this equivalence, and thus for any two equivalent sequences either both are fair or both are unfair.

Since most models of concurrency assume the fundamental liveness property of *progress*, i.e. some computation step will occur in some component of the system as long as the system is not deadlocked, including a notion of fairness into the model is reasonable only if there are legal programs in the model which feature a liveness property which could not be stated without the fairness notion. Liveness enhancement requires that every notion of fairness be necessary in order to show additional liveness properties of legal programs.

The above three criteria, however, are dependent on particular properties of the underlying computational models, and furthermore there is no widespread consensus as to whether the satisfaction of these criteria suffices for a universal definition of fairness. It is observed by [45], however, that all notions of fairness share several characteristics: first, they result in the exclusion of legal infinite execution sequences, and consider all finite execution sequences as

fair; second, they require that each component of a concurrent system execute infinitely often with an unbounded delay between executions, as long as this delay remains finite; and third, they do not affect safety properties, but do affect other liveness properties.

Interpretations of Component, Capable of Evaluating, and Sufficiently Often

A "component" can be a process (or thread), an event, a synchronisation action, a communication (or method invocation), a guard satisfaction, a transition, or a state; depending on the particular properties of the component, "capable of evaluating" is usually identified with being capable of execution or taking place: a process or thread which can execute its next instruction, an event which can occur, a synchronisation action which can be achieved, a communication channel which can receive a message, a guard which can be satisfied, a transition which can be followed, or the system can enter a state.

Interpretations of "sufficiently often" generally fall into three categories: *constraint-free*, *infinitely often*, and *almost always*. A constraint-free interpretation leads to an unconditional notion of fairness where each component of a system must proceed each time it becomes possible. An infinitely often interpretation leads to notions of fairness where components which become infinitely often possible must also proceed infinitely often. Finally, almost always interpretations require components to proceed continuously after some point in an execution.

Negative and Positive Definitions

There are two approaches to introducing fairness into computational models which do not make any provision for such constraints, termed negative and positive [45].

The negative approach usually considers two semantic "levels" [49]: the

"lower" level admits all valid executions, while the "upper" level contains mechanisms which exclude those valid executions which are unfair; an example of such a system is to introduce a proof rule for each fairness notion and then use the rules to prove termination under the constraints imposed by the fairness notions. An alternative negative approach is to extend the semantics of the underlying model with explicit fairness constraints [75, 20].

The positive approach aims at creating only fair executions, and hence deals with a single semantic level; tyically, the underlying model is modified so that fairness constraints are part of the proof rules (for example [21], which we consider later on).

The motivation for the negative approach is to produce correctness proofs with respect to a particular scheduling policy, and enforce these through implementation; the motivation for the positive approach is to produce a set of guidelines for acceptable schedulers from the point of view of the semantics of the underlying model [45].

2.3.8 Principal Fairness Notions

In [49, 75] the authors define *maximality*[3] (also called *unconditional fairness* and *impartiality*, and known as *fairmerge* for ω-regular languages [71]), *weak* fairness (also called *justice*), and *strong* fairness (also called *fairness*); *process liveness* [7] is a modification of unconditional fairness which takes into account the possiblity for processes to terminate.

The principal notions of fairness are weak and strong [72, 74, 35], which also form the foundation for further refinements resulting in notions such as *finitary*, κ*-strong*, ∞*-strong*, *equifairness*, α*-strong*, and many others [45]. In

[3]Maximality requires that each component proceed without constraints, or in other words proceed whenever it becomes enabled; this notion is too restrictive because it cannot account for components becoming disabled (such as when they cannot proceed autonomously); hence we will not deal with maximality at all.

the remainder of this section we will look at weak and strong fairness; for other approaches see [85] and [87]. Furthermore, weak and strong process, channel communication, and process communication fairness are considered for CSP in [43]. An overview of weak and strong fairness for choice is presented in [34]. An application of weak and strong fairness to events is developed in [74], and for transitions and states in [76].

Weak Fairness

Weak fairness prohibits executions where a component 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.

Because termination implies that a component is not continuously enabled, weak fairness implies process liveness. Furthermore, if a component becomes disabled only due to termination, then weak fairness coincides with process liveness. Finally, if components are continuously enabled and never terminate, then weak fairness conincides with unconditional fairness.

We consider the example of listing 2.4, adapted from [45]: two processes, A and B are executing in parallel, resulting in the system of processes P. Process A is continuously enabled and attempts to communicate with process B by sending the value 0 to B; process B initially assigns the value 1 to variable x and can then repeatedly chose between communicating with A (thus x being assigned the value 0) or[4] performing an internal action (printing the value of x if it is greater than zero). Once a communication between A and B occurs, there is no choice in the action of process B as the value of x will be 0, and the execution of P terminates.

Without any notion of fairness, the execution could always choose the sec-

[4]We will assume the keywords *or* to mean non-deterministic choice.

```
1  A ::= B!0;
2
3  B ::= x = 1;
4        while (true) {
5          A?z; x = z; print x;
6          or
7          if (x > 0) print x;
8        }
9
10 P ::= ( A || B )
```

Listing 2.4: A system of processes demonstrating the effects of weak fairness.

ond action of process B, and hence print the value 1 forever. If weak fairness is imposed, however, eventually process A must cease to be enabled, and this can happen only by choosing the first action of process B, resulting in the system printing the value 0 and terminating. Although there is no upper bound on the number of 1s printed by the system, any weakly-fair execution o P will result in a finite number of 1s followed by a 0.

Considering the example of listing 2.5, again adapted from [45], we see an inherent limitation of weak fairness, in that a process can be enabled infinitely often (though not continuously) and yet never be given the opportunity to execute. Process B has been modified so that the conditions satisfying the guard of the first action require the value of x to be odd, and hence this action is enabled only every other step in the execution (as the second action increments the value of x by one each time it executes).

In the second example, process A is disabled every other step, and hence does not remain continuously enabled, rather, it becomes enabled infinitely often; thus, the system P can result in printing the incrementing value of x forever, and remain weakly-fair. A solution to this problem is offered by strong fairness, in the next section.

```
1  A ::= B!0;
2
3  B ::= x = 1;
4       while (true) {
5         x mod 2 == 1 && A?z; x = z; print x;
6         or
7         if (x > 0) print x; x = x + 1;
8       }
9
10 P ::= ( A || B )
```

Listing 2.5: A system of processes demonstrating the limits of weak fairness.

In [21] Costa et al. develop weak (and strong, see next section) 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 in a positive way (see section 2.3.7 above) so that only weakly-fair execution sequences are admitted.

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. Because all fairness constraints are satisfied within each locally weakly-fair sequence, there is no *predictive choice*: the property of systems in which fairness constraints accumulate throughout the execution progressively limiting the choice of future actions.

Implementation costs of weak fairness are considered in [75]. A truly concurrent system will guarantee weak fairness, while a shared concurrent system will be weakly fair when a *round-robin* scheduler is employed; furthermore, a *busy-wait* implementation of a *semaphore* is weakly fair [45].

Strong Fairness

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.

Strong fairness may be more desirable than weak fairness because it solves the problem under weak fairness where some components may be ignored throughout executions. Considering the second example from the previous section, of listing 2.5, we notice that a strongly-fair execution would require the execution of process A, as it becomes enabled and disabled infinitely often; therefore, the system P always terminates after a finite number of steps under strong fairness.

In [21] Costa et al. also present strong fairness for CSS. 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.

For instance, consider the following CCS system with two processes:

$$E = fixX(a.X + b(c.NIL + d.X) \qquad F = \bar{c}.NIL$$

where the term $c.NIL$ is added for a possible execution, δ, such as the following:

$$\delta = ((E + c.NIL|F)\backslash c \xrightarrow{a} (E|F)\backslash c \xrightarrow{a} \ldots \xrightarrow{a} (E|F)\backslash c \xrightarrow{a} \ldots$$

during which the subcomponent $\bar{c}$ loses its liveness and never regains it. However, it is possible at each step in δ for $\bar{c}$ to begin regaining and losing its liveness infinitely often, as in the following example execution:

$$(E|F)\backslash c \xrightarrow{b} ((c.NIL + d.E)|F)\backslash c \xrightarrow{d} (E|F)\backslash c \xrightarrow{b} \ldots \xrightarrow{d} (E|F)\backslash c \xrightarrow{b} \ldots$$

At no step in δ is it safe, therefore, to claim that $\bar{c}$ will not become live infinitely often. In order to make such a claim we need to consider the entire future execution sequence.

Strong fairness implies weak fairness, and furthermore, if components which become disabled never become enabled again, strong fairness coincides with weak fairness [45]. An implementation of strong fairness always requires some form of queueing mechanism, such as a *priority queue* or a *first-in-first-out semaphore* [45].

2.4 Conclusions

The various contemporary object-oriented languages which feature chords as a mechanism of concurrency clearly show that the concept of a join is compatible with objects and in many cases gives rise to concise, intuitive synchronisation patterns. Chords have been implemented as language constructs, as code which is translated into an underlying language, and purely as library functions, demonstrating the flexibility of object-oriented languages in incorporating join concurrency. Furthermore, most chorded languages offer chords in addition to existing concurrency constructs, which makes chords interesting constructs both in isolation and in conjunction with other language facilities.

However, there are three important points regarding chords which remain unclear. First, there is no consensus on a definition for a chord: some languages offer synchronous communication, giving rise to many kinds of chords (synchronous, asynchronous, a combination of both), while others offer only an asynchronous core through which synchronous communication is encoded, thus providing a single kind of chord (purely asynchronous).

Second, there is no formal treatment of the semantics of chords: chorded

languages are described informally through examples, and rely on library implementations and other facilities provided by the underlying language. Furthermore, not all language constructs are employed in such examples. This may result in ambiguities, lack of consistency, and uncairtainty in how typical features of object-oriented languages, such as fields, monitors, and explicit thread life-cycles, interact with chords.

Third, all chorded languages are presented with implicit assumptions regarding the fair treatment of processes by the scheduler; furthermore, there is an underlying reliance on fairness in order to demonstrate the programming of popular synchronisation constructs (such as rendezvous, readers/writers, and so on). It is quite possible that certain language features (such as spawning new processes), or entire patterns of synchronisation constructs (such as choice between competing chords), cannot be guaranteed to function properly without explicit fairness properties.

Therefore, we identify a need to provide a featherweight model for chorded languages which offers a generalised definition of a chord, a core operational semantics of chords, and a mechanism by which fairness notions can be explicitly expressed.

Chapter 3

A Featherweight Model for Chorded Languages [1]

3.1 Introduction

We present a featherweight model of chorded languages, namely the Small Chorded Object-Oriented Language (SCHOOL). Our formalisation aims at capturing the essence of the concurrent behaviour of chords, and thus lacks many common object-oriented features which are not directly related to chords (such as fields and exceptions). In order to decide which constructs of chorded languages to include in our model, we present a derivation of a generalised chord which is similar to that of Polyphonic C$^\sharp$.

We also examine the interaction of chords with fields, a standard feature of many object-oriented languages; we find that fields are orthogonal to chords in terms of concurrent program behaviours, and can be encoded using only

[1] The work presented in this chapter has evolved from a previous model, published in [24], which was the result of collaboration with Sophia Drossopoulou, Alex Buckley, and Susan Eisenbach. The original model presented SCHOOL, a calculus of chords; the calculus in this chapter is significantly different; furthermore, work regarding the encoding of fields is original to this thesis.

chords. We extend SCHOOL by adding fields, resulting in ^{f}SCHOOL, and define an encoding between the two languages. We then prove equivalence between encoded programs and hence show that fields do not add expressive power to SCHOOL.

Structure of This Chapter

Section 3.2 presents the derivation of a general definition for a chord, along with an example on programming using this definition; section 3.3 describes SCHOOL, several of its properties including soundness and progress, and a comparison with Polyphonic C$^\sharp$, as well as some notes on a virtual machine implementation; section 3.4 presents ^{f}SCHOOL and its properties; finally, section 3.5 describes the encoding of ^{f}SCHOOL in SCHOOL, and the proof of equivalence between SCHOOL and encoded ^{f}SCHOOL programs.

3.2 Derivation of Chords

Chorded programs 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 reflects the *join* notion. 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 given chord header, however, 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*, as there is no body of expressions to result in a return value. A chord which does not contain a synchronous method is called asynchronous; such chords will not have an invoking thread blocking on them. Therefore, when an invocation for each of their participating 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 (of type *Object* in this case) is returned. The body of the chord will execute only when the chord joins, which requires an invocation of the asynchronous method `put` to be present.

When a join occurs, the invocation of `put` is consumed. Multiple invocations of `put` will result in the invocations being placed into the execution for joining later with invocations of `get` (the invocations are "queued").

Example: a Countdown Latch

Consider the following problem: an expression executing in a thread T requires the results of several other threads $R_1, \ldots, R_n$, before it can continue execution. These threads complete asynchronously, and hence T must wait for their completion (the order of completion is unimportant). One solution is to use a countdown latch; we compare a chorded implementation of such a latch (listing 3.1) with a traditional implementation (in Java, listing 3.2).

A countdown latch can be expressed as a single chord. The thread T invokes the `await` method and passes the value of `permits`, i.e. the number of other threads it wishes to wait for. Since this is a synchronous call, it will block T until the chord joins.

The threads $R_1, \ldots, R_n$, upon completion of their tasks, will signal the latch by invoking `countDown`. This invocation is asynchronous, as these threads should not have to wait for T to perform any operations.

The first time `countDown` is invoked, it will cause the chord to join, and its body will execute. The value of `permits` will be decremented by one, and `await` will be synchronously invoked again with the new value. After n joins, the value of `permits` will be zero, and the recursion will unwind, effectively unblocking T. Since invocations to `countDown` are queued, they can arrive during the execution of the chord body by T without affecting the end result.

In a Java version of the countdown latch, a field, `permits`, holds the number of other tasks the thread T must wait for. The information contained in this field is shared between the two methods (and hence between threads T and $R_1, \ldots, R_n$), and hence must be encapsulated in the latch object instead of remaining local to a method, as in the case of the chorded implementation.

The method `await` sets the initial value of `permits` and then blocks, which is achieved by invoking `wait`, available to all objects in Java. This method will return upon a `notify` invocation on the same object. Because the field

```
1 class CountdownLatch {
2   void await(int permits) & async countDown() {
3     if (permits - 1 > 0) { await(permits - 1); }
4   }
5 }
```

Listing 3.1: Chorded Countdown Latch

`permits` is shared between multiple threads, it needs to be accessed within synchronised blocks to avoid race conditions.

As a thread R_i completes and invokes `countDown`, the value of `permits` is decremented. If the value reaches zero, it means that thread R_i is in fact the last thread to notify of completion, and hence thread T must be unblocked (via invocation of `notify`).

In Java it is possible for the `wait` method to *spuriously* wake up and return. Thus, we must place the call to `wait` inside the loop `while` (`permits` $>$ `0`), which ensures the latch condition is honoured.

Finally, the semantics of `wait` are such that it must be invoked within a *synchronized* block, but upon invocation it releases the monitor acquired by this block. Then, upon waking up, it attempts to re-acquire the monitor (so there can be no race-condition in the window of time between `wait` returning and the evaluation of the loop condition, which accesses `permits`).

Comparing the two implementations, we can identify the following problematic aspects of the Java solution:

- Shared variables: can lead to race-conditions; they must have all access protected by means of mutual exclusion.
- Magic library methods: reliance on methods outside the programming language in order to perform primitive operations such as concurrency.
- Spurious notifications: programs have to include code in order to deal

```
1  class CountdownLatch {
2   int permits;
3   void  await(int permits) throws InterruptedException {
4    synchronized(this) {
5     this.permits = permits;
6     while (permits > 0) { wait(); }
7    }
8   }
9   void countDown() {
10   synchronized(this) {
11    if (--permits == 0) { notify(); }
12   }
13  }
14 }
```

Listing 3.2: Java Countdown Latch

with memory model deficiencies, increasing complexity and reducing comprehensibility.

- Irrelevant delays: threads $R_1, \ldots, R_n$ must wait for the internal logic of the countdown latch to execute every time they invoke `countDown`. The logic of the latch, however, is irrelevant to these threads.

- Invisible semantics: the interaction between `wait` and *synchronized* blocks is invisible in terms of language constructs.

The above are not specific to Java; programs in C♯ suffer from similar problems. Note that in the above example, and throughout this chapter, we will ignore issues such as restricting the use of the countdown latch object to a single thread by utilising, for instance, private fields for referring to the object; we will also ignore usage rules, such as ensuring a constructor is called so that the object is not null.

3.3 SCHOOL

SCHOOL 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 overridden.

The generalised chord derivation of the previous section closely resembles the chords of Polyphonic C♯, which features the *async* return type, a special subtype of *void*, to indicate asynchronous methods; this return type is not, however, necessary in order to implement asynchronous methods once we observe that all methods of return type *void* can be executed asynchronously, as the invoker is not expecting a return value.

The chords of SCHOOL are similar to those presented in section 3.2, however, SCHOOL does not feature the *async* type; instead, a method which has a return type of *void* can be invoked either synchronously or asynchronously, depending on whether this method is found in the synchronous or asynchronous part of a chord header. SCHOOL programs have potentially more valid behaviours than chords with explicit *async* return types; hence, SCHOOL constitutes a more general model of chorded behaviours than existing experimental languages.

Furthermore we require exactly one argument for each method, as parameter passing is not particularly interesting in the context of chords. We name the argument of method `m` as `m_x`. The value of the last expression evaluated in a body becomes the return value of the chord.

3.3.1 Abstract Syntax and Program Representation

Figure 3.1 provides an overview of syntax and program representation. The syntax of SCHOOL expressions, $Expr$, consists of method calls $e.m(e)$, variables x, object creation $new\ c$, the special receiver $this$, and the $null$ value.

Abstract Syntax

$$
\begin{aligned}
e \in Expr &::= null \mid this \mid x \mid new\ c \mid e.m(e) \mid e;e \\
MethSig &::= t\ m(c) \\
t \in Type &::= void \mid c \\
x, c, m \in Id &
\end{aligned}
$$

Program Representation

$$
\begin{aligned}
Program &= (Id^c \to Id^c) \times (Id^c \times Id^m \to MethSig) \times (Id^c \to \mathcal{P}(Chord)) \\
Chord &= (Id^m \cup \{\varepsilon\}) \times \mathcal{P}(Id^m) \times Expr
\end{aligned}
$$

Runtime Entities

$$
\begin{aligned}
Configuration &= RExpr^\star \times Heap \\
Heap &= \mathbb{N} \to Id^c \\
re \in RExpr &::= voidValue \mid null \mid \iota \mid new\ c \mid re.m(re) \mid re;re \\
\iota \in \mathbb{N} &
\end{aligned}
$$

Figure 3.1: SCHOOL overview.

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. SCHOOL 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 another class name: $Id^c \to Id^c$, expressing the direct superclass relationship between the classes; 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: $Id^c \times Id^m \to MethSig$; in order to look up the method signature for method m of class c we use $P{\downarrow_2}(c,\ m)$. The signature itself consists of a return type, a name, and the class type of the single argument: $t\ m(c)$.

$$P{\downarrow_1}(CountdownLatch) = Object$$
$$P{\downarrow_2}(CountdownLatch, await) = void\ await(int)$$
$$P{\downarrow_2}(CountdownLatch, countDown) = void\ countDown(Object)$$
$$P{\downarrow_3}(CountdownLatch) = \{(await, \{countDown\}, if(await\_x - 1 > 0)this.await(await\_x - 1);)\}$$

Figure 3.2: SCHOOL representation of the Countdown Latch program.

The third component maps a class name to the set of chords defined in the class: $Id^c \rightarrow \mathcal{P}(Chord)$; 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 (indicating that the chord lacks a synchronous method and hence is asynchronous). The second element is a set of asynchronous method names. The third is the body of the chord.

The program of listing 3.1 is represented in abstract syntax as in figure 3.2, ignoring the rule about number of parameters, and assuming that arithmetic and integers are defined in the language.

3.3.2 Execution

Execution of SCHOOL 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:

$$e_1, \ldots, e_n, h \quad \longrightarrow \quad e'_1, \ldots, e'_m, h'$$

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. Hence it is always the case that $m \geq n$.

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

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

Since $\overline{e}$ denotes concurrency not sequentiality, expressions in $\overline{e}$ are separated by "," instead of ";".

We describe SCHOOL using a structural operational semantics found in figure 3.3; 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; the result is 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 $E[\bullet] \neq [\bullet]$, requiring the evaluation context to not be empty, is necessary so that we avoid infinite reductions of the form:

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

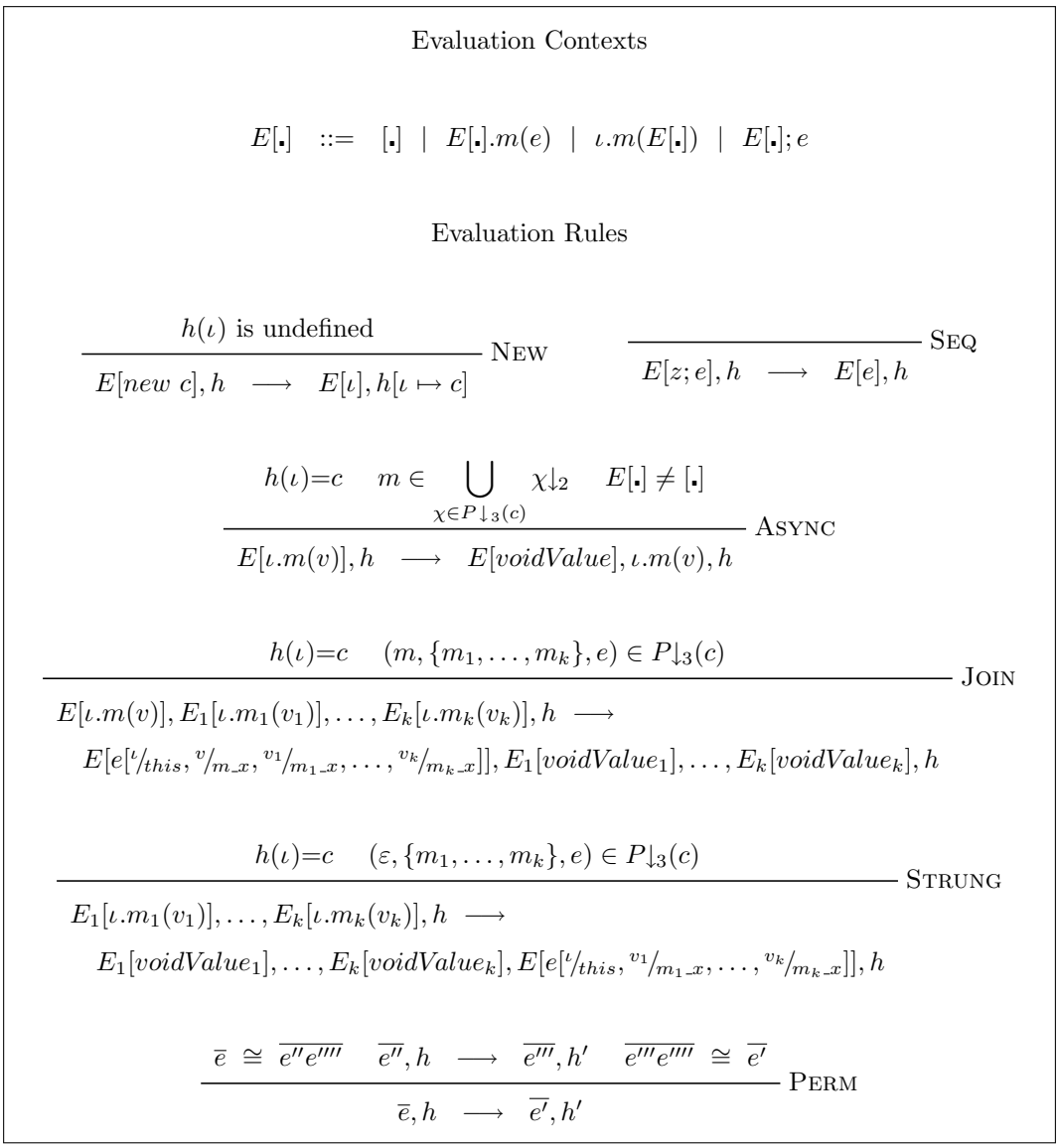

Figure 3.3: SCHOOL operational semantics.

- Join: selects a chord in which a synchronous method participates and *joins* this chord by 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,

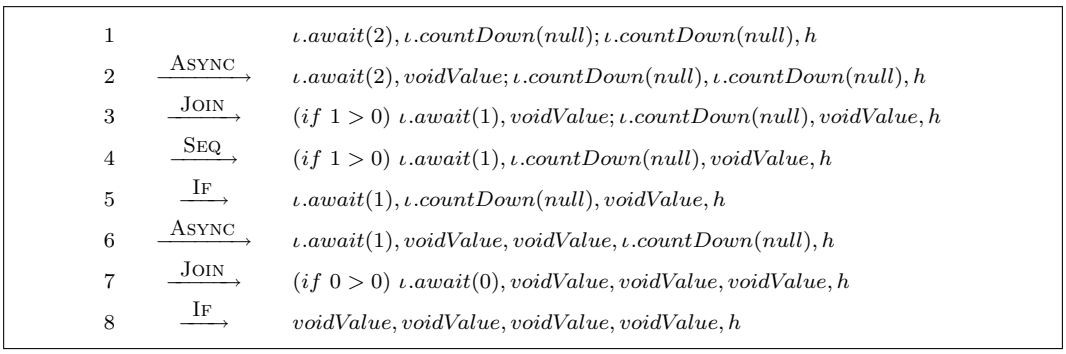

Figure 3.4: Example of the Countdown Latch execution in SCHOOL.

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 $\overline{e} \cong \overline{e'}$ means that $\overline{e'}$ is a permutation of $\overline{e}$.

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.

For example, consider two threads using the countdown latch of figure 3.2: one thread invokes `await( 2 )` and another thread invokes `countDown`() twice. We assume that the heap h has an instantiated countdown latch at address ι, and we assume standard operational semantics for sequential execution and if-statements; an execution sequence is in figure 3.4. Because this example requires a conditional rule for the if-statement, which is not part of SCHOOL as we have focused on the concurrent aspect of chords, we show this rule in figure 3.5 for the sake of the example; another option would be to encode the if-statement using methods. We will not deal with the commonly employed *else* part, and we will assume booleans have been defined in the language.

On line 1 we start with the two threads. On line 2 the first invocation of `countDown` from the second thread is evaluated through the ASYNC rule and

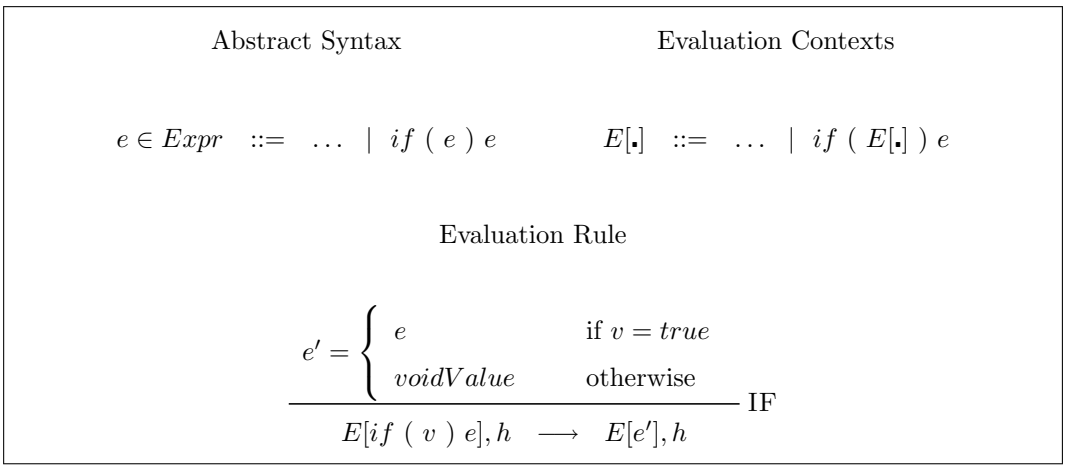

Figure 3.5: SCHOOL if-statements.

results in the invocation being replaced with *voidValue* and the invocation itself being placed into the execution as a new thread. On line 3 the first thread's invocation of `await` joins with the newly available invocation: the body of the chord is the currently evaluating expression for the first thread and *voidValue* is the value of the third thread. On line 4 the *voidValue* of the second thread is removed since it is sequential composition, and on line 5 the condition in the if-statement is evaluated. In the next few lines similar executions take place, but on line 8 the condition is false and the chord executing in the first thread eventually returns.

The heaps of SCHOOL are particularly simple: they only record the existence of objects and their associated type, and serve only as a means of storing objects and allowing method invocations to determine their receivers. This is possible because SCHOOL lacks many of the object oriented features that require a more structured heap. In particular, queues of messages to asynchronous methods are implicit and are determined by examination of the configuration. If this were not possible, then explicit queues of method invocations would have to be encoded, per object, in the heap, and the language semantics would be significantly more complex.

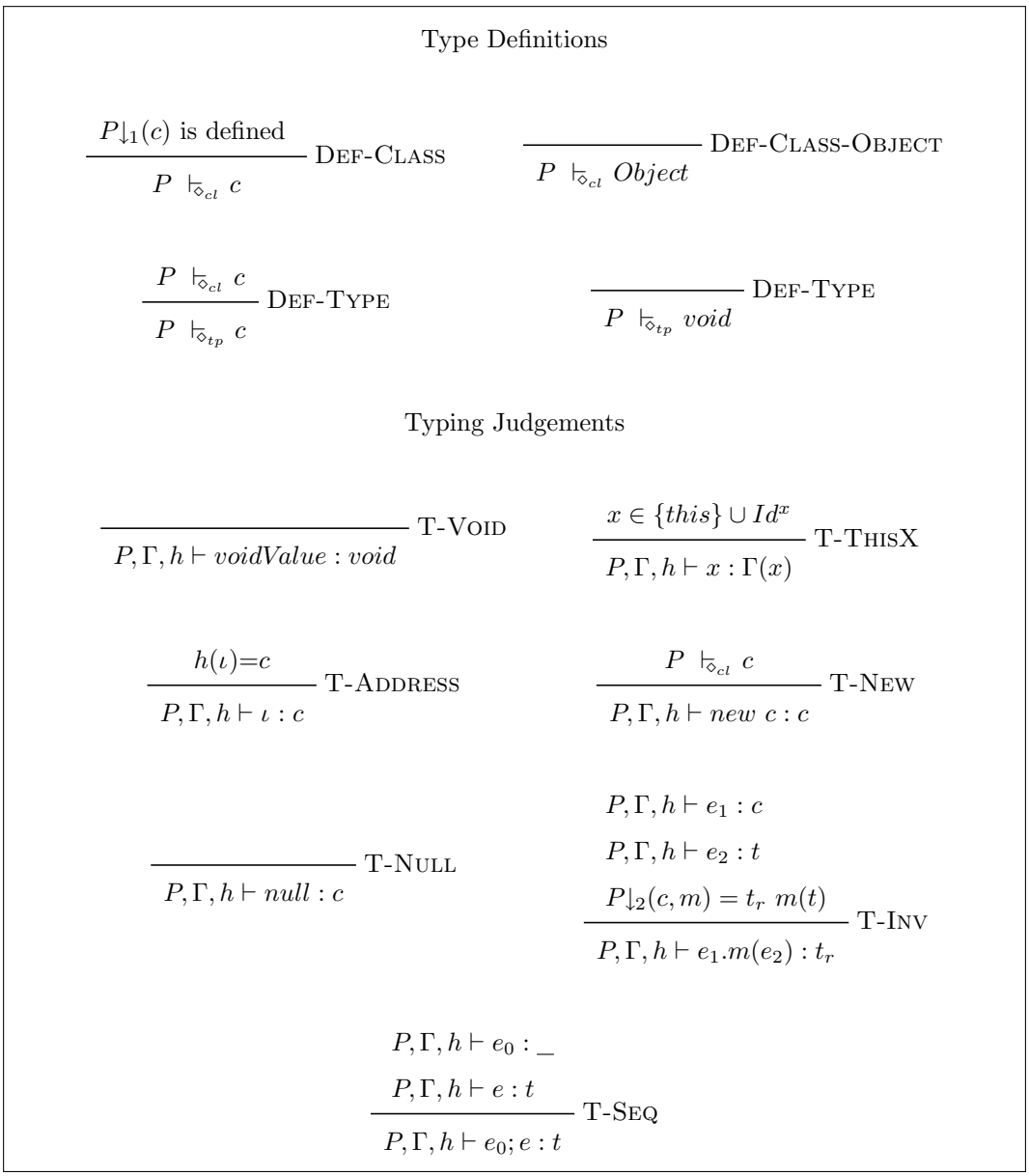

Figure 3.6: SCHOOL type system.

3.3.3 Types, Well-Formedness and Subject Reduction

The type system of SCHOOL (presented in figure 3.6) is nominal, with class names constituting types. There is an implicit empty top superclass called *Object*, and all other class types are defined through class declarations (we write $P \ \vdash_{\diamond_{cl}} c$ to state that class c is declared in program P). In addition to classes, there is a primitive type, *void*. We write $P \ \vdash_{\diamond_{tp}} t$ to indicate that

t is a type in program P. Terms in SCHOOL are typed using a context, Γ, and a heap, h. We use the same type system for source level and runtime expressions.

For a class c of a program P to be well-formed, denoted $P \vdash c$ (see definition 1), it is necessary to fulfil four criteria:

1. It must have a superclass in P.

2. All method signatures appearing in its superclass must also appear in the class.

3. Each method signature must participate in at least one chord; if the method's return type is *not void* then the method signature must participate as the synchronous part of a chord.

4. For each chord appearing in the class, all of its participating methods must be present; furthermore, the asynchronous methods (if any) must have a return type of *void*; finally, the type of a chord body must agree with the return type of the synchronous method, or be *void* if the chord is asynchronous.

Definition 1 (SCHOOL Well-Formed Classes)

$P \vdash c$ iff :

1. $P \ \vdash_{\diamond_{cl}} P{\downarrow_1}(c)$
2. $P{\downarrow_2}(P{\downarrow_1}(c), m)$ defined $\implies P{\downarrow_2}(c, m) = P{\downarrow_2}(P{\downarrow_1}(c), m)$
3. $P{\downarrow_2}(c, m) = t\ m(\_) \implies \begin{cases} m \in \bigcup_{\chi \in P\downarrow_3(c)} \{\chi{\downarrow_1}\} \cup \chi{\downarrow_2} & \text{if } t = \mathit{void} \\ m \in \bigcup_{\chi \in P\downarrow_3(c)} \{\chi{\downarrow_1}\} & \text{otherwise} \end{cases}$
4. $(\alpha, \{m_1, \ldots, m_n\}, e) \in P{\downarrow_3}(c) \implies$

   $\forall\ i \in 1..n\ :\ \exists\ t_i\ :\ P{\downarrow_2}(c, m_i) = \mathit{void}\ m_i(t_i)$

   $P, m_1\_x \mapsto t_1, \ldots, m_n\_x \mapsto t_n, \mathit{this} \mapsto c, \gamma, \_ \vdash e : t$

   where, either $\alpha = \varepsilon,\ \gamma = \varepsilon,\ t = \mathit{void}$

   or $\alpha = m,\ P{\downarrow_2}(c, m) = t\ m(t'),\ \gamma = m\_x \mapsto t'$

A program P is well-formed, if all declared classes are well-formed.

Definition 2 (SCHOOL Well-Formed Programs)

$\vdash P$ iff $P \ \vdash_{\diamond_{cl}} c \implies P \vdash c$

Subject reduction (theorem 1) guarantees for a well-formed program P, a heap h, and well-typed expressions $\overline{e}$, execution results in a heap h' and well-typed expressions $\overline{e'}$; furthermore, the type of each expression remains unchanged, and if the evaluation spawned a new thread, then this thread is well-typed too (with type *void*). The theorem relies on appropriate substitution of variables during method invocation (definition 3) and three auxiliary lemmas: substitution preserves types (lemma 1), evaluation preserves the types of objects in the heap (lemma 2), and the evaluation of a single expression preserves its type (lemma 3).

Definition 3 (Substitution of Variables)

For a substitution $\sigma = Id \cup \{\mathit{this}\} \to \mathit{Addr}$*, a heap* h*, and an environment* Γ*, we have:* $P, \Gamma, h \vdash \sigma$ iff $\forall\ \mathit{id} \in \mathit{dom}(\Gamma)\ :\ P, \Gamma, h \vdash \sigma(\mathit{id}) : \Gamma(\mathit{id})$

Lemma 1 (Substitution Preserves Types)

$$\left.\begin{array}{l} P, \Gamma, h \vdash e : t \\ P, \Gamma, h \vdash \sigma \end{array}\right\} \implies P, \Gamma, h \vdash [e]_\sigma : t$$

Proof *By structural induction on the derivation of* $P, \Gamma, h \vdash e : t$*; case analysis on the typing judgement used, and use of definition 3. The cases for* T-Void, T-Null, T-New, *and* T-Address *are immediate as they do not involve variables. The cases for* T-Seq, T-Subclass, *and* T-Inv *are satisfied inductively. Finally, in the case for* T-ThisX *we observe from the first premise of the lemma,* $P, \Gamma, h \vdash e : t$*, and the conclusion of the typing judgement that* e *is of the form* x *and hence* $P, \Gamma, h \vdash x : \Gamma(x)$*, so* $P, \Gamma, h \vdash e : \Gamma(x)$ *where* $t = \Gamma(x)$ *; we also observe from the second premise of the lemma,* $P, \Gamma, h \vdash \sigma$*, and the definition of* σ*, that* $P, \Gamma, h \vdash \sigma(id) : \Gamma(id)$*, and therefore* $P, \Gamma, h \vdash \sigma(x) : \Gamma(x)$*; we conclude that* $P, \Gamma, h \vdash [e]_\sigma : \Gamma(x)$*, which satisfies the lemma.* □

Lemma 2 (Evaluation Preserves Types of Objects in the Heap)

$$e, h \longrightarrow e', h' \implies \forall\ \iota \in dom(h)\ :\ h(\iota) = h'(\iota)$$

Proof *By case analysis on the evaluation rule used in the derivation of* $e, h \longrightarrow e', h'$*. The only applicable rules are* New *and* Seq*. For the case of* New *we obtain from its premise that* $h(\iota)$ *is undefined, while from its conclusions we obtain* $h[\iota \mapsto c]$*; we therefore conclude that* $\forall\ \iota \in dom(h)\ :\ h(\iota) = h'(\iota)$*, which satisfies the lemma. For the case of* Seq *we obtain that* $h = h'$*, which immediately satisfies the lemma.* □

Lemma 3 (Evaluation of Single Expression Preserves its Type)

$$\left.\begin{array}{l} \vdash P \\ P, \Gamma, h \vdash e : t \\ e, h \longrightarrow e', h' \end{array}\right\} \implies P, \Gamma, h' \vdash e' : t$$

Proof *By structural induction on the derivation of* $e, h \longrightarrow e', h'$*; case analysis on the last evaluation rule applied. The only applicable rules are* New *and* Seq.

For the case of New *we obtain from its premise that* $h(\iota)$ *is undefined, while from its conclusions we obtain that the form of* e *is new* c*, and that of* e' *is* ι*, as well as that the resulting heap is* $h' = h[\iota \mapsto c]$*; from the second premise of the lemma and the form of* e *we can apply the typing judgement* T-New *(the first premise of the lemma satisfies the premise of this typing judgement), and obtain* $P, \Gamma, h \vdash new\ c : c$*, and hence* $t = c$*; from the form of* e' *and the resulting heap* h' *we can apply the typing judgement* T-Address *and conclude that* $P, \Gamma, h' \vdash \iota : c$*, which satisfies the lemma.*

For the case of Seq *we obtain that the form of* e *is* $z; e_0$ *for some expression* e_0*, that the form of* e' *is* e_0*, and that* $h = h'$*; from the second premise of this lemma and the form of* e *we can apply the typing judgement* T-Seq *and obtain* $P, \Gamma, h \vdash e : t'$ *so* $t = t'$*; from the form of* e' *and the fact that the heap remains unchanged we conclude that* $P, \Gamma, h' \vdash e' : t'$*, which satisfies the lemma.* □

Theorem 1 (SCHOOL Subject Reduction)

$$\left.\begin{array}{l} \vdash P \\ e_1, \ldots, e_n, h \longrightarrow e'_1, \ldots, e'_n, e, h' \\ \forall\ i \in 1..n\ :\ P, \Gamma, h \vdash e_i : t_i \end{array}\right\} \implies \begin{array}{r} \forall\ i \in 1..n\ :\ P, \Gamma, h' \vdash e'_i : t_i \\ \wedge\ \ P, \Gamma, h' \vdash e : void\ \text{ if }\ e \neq \varepsilon \end{array}$$

Proof *By structural induction on the derivation of* $e_1, \ldots, e_n, h \longrightarrow e'_1, \ldots, e'_n, e, h'$*; case analysis on the last evaluation rule applied, and use of definition 3 and lemmas 1, 2, and 3. In the following cases we will use the shorthand notion* $\overline{e} \equiv e_1, \ldots, e_n$ *and* $\overline{e'} \equiv e'_1, \ldots, e'_n, e$*.*

—

The case for Perm *is satisfied inductively, where we appeal to lemma 3 in the case of a single expression evaluating (through* New *or* Seq*), or the*

*inductive hypothesis of this theorem in all other cases (*ASYNC, JOIN, *and* STRUNG*). Furthermore, in each of the three cases the heap remains unchanged, and using the above result that evaluation does not change the types of objects in the heap (lemma 2), we immediately obtain that the type of an expression in the initial configuration remains unchanged in the final configuration when the expression does not evaluate.*

—

In the case for ASYNC *we observe that the form of* $\overline{e}$ *is* $E[\iota.m(v)]$ *and that of* $\overline{e'}$ *is* $E[voidValue], \iota.m(v)$*, while the heap remains unchanged, hence* $h' = h$*; from the premises we know that the receiver object is of type* c*, since* $h(\iota)$=c*, and we know that the method* m *participates in the asynchronous part of at least one chord, since* $m \in \bigcup_{\chi \in P\downarrow_3(c)} \chi\downarrow_2$.

Since $\overline{e}$ *is well-typed (by the second premise of the theorem), from the form of* $\overline{e}$ *we can apply the typing judgement* T-INV *and obtain (a)* $P, \Gamma, h \vdash \iota.m(v) : t_r$ *for some type* t_r*, (b)* $P, \Gamma, h \vdash \iota : c'$ *for some class* c'*, (c)* $P, \Gamma, h \vdash v : t_v$ *for some type* t_v*, and (d)* $P\downarrow_2(c', m) = t_r\ m(t_v)$.

Since $h(\iota)$=c*, from (b) we obtain that* $c' = c$*, and subsequently from (d) we obtain* $P\downarrow_2(c, m) = t_r\ m(t_v)$.

Since the method m *participates asynchronously in at least one chord, and since the program is well-formed (from the first premise of the theorem), we refer to well-formedness and obtain* $P\downarrow_2(c, m) = void\ m(\_)$*, which immediately gives us that* $t_r = void$ (t_v *from (c) is consistent with this). Using our knowledge of* t_r *and referring back to (a), we obtain* $P, \Gamma, h \vdash \iota.m(v) : void$*; furthermore, using the fact that the heap remains unchanged, we obtain* $P, \Gamma, h' \vdash \iota.m(v) : void$.

From the form of $E[voidValue]$ *and from the fact that the heap remains unchanged, we can apply the typing judgement* T-VOID *and obtain* $P, \Gamma, h' \vdash E[voidValue] : void$.

Therefore, the single expression from $\overline{e}$ did not change type, and the new expression created in $\overline{e'}$ is of type void*, which satisfies the theorem.*

—

In the case for Join *we observe that the form of $\overline{e}$ is $E[\iota.m(v)], E_1[\iota.m_1(v_1)], \ldots, E_k[\iota.m_k(v_k)]$ and that of $\overline{e'}$ is $E[[e]_\sigma], E_1[voidValue_1], \ldots, E_k[voidValue_k]$, where we have a substitution $\sigma = [{}^{\iota}/_{this}, {}^{v}/_{m\_x}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]$, while the heap remains unchanged, hence $h' = h$; from the premises we know that the receiver object is of type c, since $h(\iota)$=c, and we know that the method m participates in the synchronous part of at least one chord, as well as that the asynchronous methods $m_1, \ldots, m_k$ form the chord's asynchronous part, since $(m, \{m_1, \ldots, m_k\}, e) \in P{\downarrow_3}(c)$.*

From the chord $(m, \{m_1, \ldots, m_k\}, e)$ and well-formedness we know that $\exists\, t, t' \;:\; P{\downarrow_2}(c, m) = t\ m(t')$ and $\forall\ i \in 1..k \;:\; \exists\ t_i \;:\; P{\downarrow_2}(c, m_i) = void\ m_i(t_i)$, and $P, m\_x \mapsto t', m_1\_x \mapsto t_1, \ldots, m_n\_x \mapsto t_k, this \mapsto c, \_ \vdash e : t$. From the typing judgement T-Address *and knowing $h(\iota)$=c, we obtain $P, \Gamma, h \vdash \iota : c$ (and consequently $P, \Gamma, h \vdash this : c$ through* T-ThisX*); and then from the typing judgement* T-Inv *we obtain $P, \Gamma, h \vdash \iota.m(v) : t \ \wedge\ P, \Gamma, h \vdash v : t'$ and $\forall\ i \in 1..k \;:\; P, \Gamma, h \vdash \iota.m_i(v_i) : void \ \wedge\ P, \Gamma, h \vdash v_i : t_i$.*

From the above we notice that for all identifiers in the substitution σ the type of the original identifier matches the type of the substituted identifier, and hence by definition 3 we conclude that $P, \Gamma, h \vdash \sigma$, and therefore by lemma 1 and the fact that the heap remains unchanged we obtain $P, \Gamma, h' \vdash [e]_\sigma : t$.

From the typing judgement T-Void *and the form of $e'_1, \ldots, e'_k$ we obtain $\forall\ i \in 1..k \;:\; P, \Gamma, h' \vdash voidValue_i : void$.*

Therefore, all expressions have retained their type, which satisfies the theorem.

—

In the case for STRUNG *we observe that the form of* $\overline{e}$ *is* $E_1[\iota.m_1(v_1)], \ldots, E_k[\iota.m_k(v_k)]$ *and that of* $\overline{e'}$ *is* $E_1[voidValue_1], \ldots, E_k[voidValue_k], E[[e]_\sigma]$*, where we have a substitution* $\sigma = [{}^{\iota}/_{this}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]$*, while the heap remains unchanged, hence* $h' = h$*; from the premises we know that the receiver object is of type c, since* $h(\iota)$=*c, and we know that the asynchronous methods* $m_1, \ldots, m_k$ *form a chord's asynchronous part, since* $(\varepsilon, \{m_1, \ldots, m_k\}, e) \in P{\downarrow_3}(c)$*.*

From the chord $(\varepsilon, \{m_1, \ldots, m_k\}, e)$ *and well-formedness we know that* $\forall\ i \in 1..k\ :\ \exists\ t_i\ :\ P{\downarrow_2}(c, m_i) = void\ m_i(t_i)$*, and* $P, m_1\_x \mapsto t_1, \ldots, m_n\_x \mapsto t_k, this \mapsto c, \_ \vdash e : void$*. From the typing judgement* T-ADDRESS *and knowing* $h(\iota)$=*c, we obtain* $P, \Gamma, h \vdash \iota : c$ *(and consequently* $P, \Gamma, h \vdash this : c$ *through* T-THISX*); and then from the typing judgement* T-INV *we obtain and* $\forall\ i \in 1..k\ :\ P, \Gamma, h \vdash \iota.m_i(v_i) : void\ \wedge\ P, \Gamma, h \vdash v_i : t_i$*.*

From the above we notice that for all identifiers in the substitution σ *the type of the original identifier matches the type of the substituted identifier, and hence by definition 3 we conclude that* $P, \Gamma, h \vdash \sigma$*, and therefore by lemma 1 and the fact that the heap remains unchanged we obtain* $P, \Gamma, h' \vdash [e]_\sigma : void$*.*

From the typing judgement T-VOID *and the form of* $e'_1, \ldots, e'_k$ *we obtain* $\forall\ i \in 1..k\ :\ P, \Gamma, h' \vdash voidValue_i : void$*.*

Therefore, all expressions have retained their type, and the new expression created in $\overline{e'}$ *is of type* void*, which satisfies the theorem.* □

Because we have focused on the concurrent aspect of chorded languages, we have not dealt with inheritance in SCHOOL beyond a rudimentary sub-type relation for classes. Well-formedness of classes requires that all method signatures appearing in a super class also appear in a sub class, which means that a given class will contain definitions for all the methods appearing in its entire class hierarchy; furthermore, since all defined methods must appear in at least one chord, all sub classes must either "copy" the chords of their super class, or "override" those chords (as long as all methods are used). The consequence

for subject reduction is that all method and chord look-up operations always succeed for well-formed programs, which greatly simplifies the type system. As overviewed earlier on (section 2.2.1), inheritance for join patterns has been thoroughly studied and numerous rich type systems have been developed.

The heaps of SCHOOL have a minimal impact on properties of the language such as subject reduction; specifically, the only evaluation rule which modifies the heap is NEW, which results in a monotonically increasing heap where no previous object is affected. The typing judgement T-ADDRESS is the only judgement which involves the heap and determines the type of an object in order to look-up methods and chords of its class. The rest of the evaluation rules leave the heap unchanged, which greatly simplifies the above proof as all expressions which do not change in the current evaluation step can immediately be shown to retain their type.

Some care must be taken when performing substitution during chord joining in SCHOOL; in contrast to traditional languages, where a method signature contains all the information necessary for substitution upon invocation, in SCHOOL a synchronous method can participate in potentially multiple chords which consist of different patterns of asynchronous methods. Hence, the same synchronous method may receive different substitutions depending on which chord it joins. The fourth part of well-formedness for classes ensures that the appropriate substitution takes place and the proof of subject reduction appeals to this construction in order to show that all valid chord look-ups satisfy the constraints of substitution.

3.3.4 Progress

Progress for SCHOOL (see theorem 2 below) is stated as follows: either a configuration can evaluate through application of an evaluation rule, or no evaluation rule is applicable and the configuration is then either *terminated* or *blocked*. Termination occurs when all expressions are *ground values* (or *null-*

pointers, see below), while a blocked configuration is one in which at least one expression is blocked, and no evaluation rule is applicable (the rest of the expressions are either blocked or grounded (or null-pointers)).

Blocked Expressions

An expression is *blocked* (see definition 4 below) if it is not a ground value or null-pointer and it is not possible to apply an evaluation rule to further evaluate it. In SCHOOL, the only such form of expression is an invocation on a non-null receiver. If the method invocation is asynchronous and in a non-empty evaluation context, it can always be further evaluated through the ASYNC rule and hence is never blocked (part 1 of the definition). If the method participates synchronously in a chord, then at least one asynchronous method invocation necessary for the joining of that chord must be missing from the configuration (part 2 of the definition). If the method participates asynchronously in a purely asynchronous chord, then at least one *other* asynchronous method invocation necessary for that chord must be missing from the configuration (part 3 of the definition). If the method participates asynchronously in a synchronous chord it is never considered blocked and is not captured by this definition (we will consider the invocation a ground value instead if the chord cannot join, see below).

Definition 4 (SCHOOL Blocked Expressions)

For a program P, a configuration $\overline{e}, h$, an expression $e \in \overline{e}$ is blocked iff it is of the form $\iota.m(\_)$ where $h(\iota)$=c and additionally and the following three hold:

1. $$m \in \bigcup_{\chi \in P\downarrow_3(c)} \chi\downarrow_2 \implies E[\blacksquare] = [\blacksquare]$$
2. $$\forall\ (m, \{\overline{m}\}, \_) \in P\downarrow_3(c)\,, \overline{m} \neq \emptyset\ :\ \exists\ m' \in \overline{m}\ :\ \iota.m'(\_) \notin \overline{e} \setminus e$$
3. $$\forall\ (\varepsilon, \{\overline{m}\}, \_) \in P\downarrow_3(c)\,, m \in \overline{m}, \overline{m} \setminus m \neq \emptyset\ :\ \exists\ m' \in \overline{m}\ :\ \iota.m'(\_) \notin \overline{e}$$

From the above definition we notice that if a method participates asynchronously in any chord and its context is not empty, then the evaluation rule Async is immediately applicable and thus the configuration is not blocked (hence part 1 requires $E[\bullet] = [\bullet]$).

Furthermore, if a chord consists of a single method, either synchronous (a degenerate chord) or asynchronous (a trivial chord), then it does not cause the configuration to be blocked, as we can immediately apply either the Join rule or the Strung rule, respectively (hence parts 2 and 3 require $\overline{m} \neq \emptyset$ and $\overline{m} \setminus m \neq \emptyset$ respectively).

Finally, if a method participates both as the synchronous and as an (or the) asynchronous method of a chord, at least *two* invocations of the method must be present (required in order to apply the Join rule), otherwise the configuration is blocked (hence part 2 requires the missing invocation to be from the set $\overline{e} \setminus e$).

Because parts 2 and 3 quantify over *all* chords, an invocation will indeed only be considered blocked when no evaluation rule can be applied to further evaluate it.

Ground Values

An expression is a *ground value* (see definition 5 below) when its form is either *voidValue*, or ι, or *null*, as there is no rule through which it can further evaluate. However, it is not obvious whether an asynchronous method invocation, $\iota.m(v)$, placed in the configuration through the Async rule, and thus having an empty context, $E[\bullet] = [\bullet]$, constitutes a ground value or is considered blocked when it cannot participate in any evaluation.

This problem can be stated as the question: is a configuration, where at least one chord's join is partially satisfied (by the presence of a subset of method invocations), blocked? It turns out that the answer is a matter of preference, as both options lead to the same theorem of progress (the proof of the theorem,

however, *is* affected by the choice of answer).

We have decided to consider irreducible asynchronous method invocations as ground values when they can only partially satisfy the join of a synchronous chord, and blocked when they can only partially satisfy the join of an asynchronous chord. The justification for this choice is that in the case of synchronous chords, emphasis is placed on the synchronous method, and therefore it is the absence of a synchronous invocation which results in a blocked execution; on the other hand, for asynchronous chords, all participating methods are equally important, and hence the absence of any invocation renders the configuration blocked.

Therefore, an asynchronous invocation is considered a *ground value* when it is an asynchronous invocation with an empty context, so that it cannot immediately evaluate through Async (part 1 of the definition), it does not participate as a synchronous method of any chord, as then it would be considered blocked not ground (part 2 of the definition), it does not participate in any purely asynchronous chords, as again it would be considered blocked not ground (part 3 of the definition), and finally whenever it participates in the asynchronous part of a synchronous chord, it is considered ground only when the chord is partially satisfied: either an invocation of the synchronous method of the chord must be missing, and/or an invocation of some other asynchronous method of the chord must be missing (part 4 of the definition).

Definition 5 (SCHOOL Ground Values)

For a program P and a configuration $\overline{e}, h$, an expression $e \in \overline{e}$ is a ground value iff it is of the form voidValue, or ι, or null; or of the form $\iota.m(\_)$ where $h(\iota)$=c and additionally the following four hold:

1. $m \in \bigcup_{\chi \in P\downarrow_3(c)} \chi\downarrow_2 \implies E[\blacksquare] = [\blacksquare]$
2. $\nexists (m, \_, \_) \in P\downarrow_3(c)$
3. $\nexists\ (\varepsilon, \overline{m}, \_) \in P\downarrow_3(c)\ :\ m \in \overline{m}$
4. $\forall\ (m', \overline{m}, \_) \in P\downarrow_3(c)\,, m' \neq \varepsilon, m \in \overline{m}\ :$
   $\iota.m'(\_) \notin \overline{e} \quad \wedge \quad \exists\ m'' \in \overline{m}\ :\ \iota.m''(\_) \notin \overline{e}$

Because part 4 quantifies over *all* synchronous chords in which the method participates asynchronously, the invocation will be considered ground only when no synchronous chord in which it participates is fully satisfied. Therefore, if the asynchronous invocation fully satisfies one chord and partially satisfies another, it is not considered ground.

Null-Pointers

Considering the case of a method invocation on a *null* address, of the form $null.m(\_)$; we notice that this expression can type check, however, it cannot reduce as there is no evaluation rule which accepts *null* in place of an ι. Therefore, we consider such invocations as *null-pointer expressions*, and in terms of progress they are treated as ground values. Whether a configuration with a null-pointer expression can be considered terminated (as would a configuration consisting of only ground values) is subject to interpretation of the meaning of termination one wishes to give.

Progress

Using the above definitions can show progress for all well-typed SCHOOL configurations: a well-typed configuration can either be further evaluated, or it consists of ground values (or null-pointers) only (has terminated), or it is blocked (at least one blocked expression when no evaluation rule is applicable and the rest of the expressions are either blocked or ground (or null-pointers)).

Theorem 2 (SCHOOL Progress)

$$\left.\begin{array}{l} \vdash P \\ P, \_, h \vdash \overline{e} : \overline{t} \end{array}\right\} \implies \left\{\begin{array}{ll} & \exists\ \overline{e'}, h' \ :\ \overline{e}, h \ \longrightarrow\ \overline{e'}, h' \\ \vee & \overline{e} \text{ consists of ground values or} \\ & \text{null-pointer expressions only} \\ \vee & \overline{e}, h \text{ is blocked} \end{array}\right.$$

Proof *By case analysis on the applicability of evaluation rules; when no evaluation rule can be applied we perform a case analysis on the participation of asynchronous methods in chords and use definitions 4 and 5 to determine whether they are ground values or whether they are blocked.* □

3.3.5 Comparison Between SCHOOL and Polyphonic C♯

There are several structural differences between Polyphonic C♯ chords and SCHOOL chords:

- While Polyphonic C♯ has methods and chords, we unify the two and treat Polyphonic C♯ methods as SCHOOL chords with a single synchronous method in their header.
- We have no *async* type, and allow a method of return type *void* to appear in synchronous and asynchronous parts of chord headers.
- In Polyphonic C♯ asynchronous methods can only be overridden by asynchronous methods, and synchronous methods can only be overridden by synchronous methods; SCHOOL does not have this restriction.
- Similarly, in Polyphonic C♯ methods are either synchronous or asynchronous; SCHOOL does not have this restriction either.
- Polyphonic C♯ class hierarchies are acyclic, while this is not a constraint imposed by SCHOOL.

The last three Polyphonic C♯ constraints are not necessary for type soundness, and so have not been included in SCHOOL.

3.3.6 Implementation

We have implemented SCHOOL in the form of a virtual machine named Harp, similar to a Java virtual machine (programs are compiled into classes with interpretable bytecodes). The source code for Harp and its compiler, along with example input programs which include the unbounded buffer and the countdown latch can be obtained from the web at the following site:

`slurp.doc.ic.ac.uk/chords/harp/`

The original design of the compiler and virtual machine, along with a prototype implementation, was by the author; subsequent refinement and new implementations were by Volanakis [86] and Nicolaou [66] as part of final year undergraduate projects supervised by the author in the Department of Computing, Imperial College London, United Kingdom.

3.4 SCHOOL With Fields: ^{f}SCHOOL

In order to focus on chords, in SCHOOL we have omitted fields. However, the functionality of fields can be obtained using only chords, and hence adding fields to SCHOOL would not increase its expressivity. To show this, we extend SCHOOL with field declarations, obtaining ^{f}SCHOOL, and then define an encoding that maps ^{f}SCHOOL programs into SCHOOL programs, and show that this encoding preserves all observable behaviours. For the remainder of the chapter we use the annotation $^{f}\ldots$ to denote ^{f}SCHOOL entities.

Listing 3.3 shows the Chorded Countdown Latch example using fields. In the original example (listing 3.1), the value of `permits` was passed in (recursive) calls to `await`(). Now, we just store its value directly in a field, which a client must initialise before calling `await`().

Hence, a class in ^{f}SCHOOL contains chord definitions as in SCHOOL, plus zero or more field declarations. Expressions in chord bodies can read from and

```
class CountdownLatch {
 int permits;
 void await( ) & async countDown( ) {
  permits = permits - 1;
  if (permits > 0) { await( ); }
 }
}
```

Listing 3.3: Chorded Countdown Latch with fields.

write to an object's fields using the dot operator.

3.4.1 Abstract Syntax and Program Representation

In order to obtain ^{f}SCHOOL we extend SCHOOL in the following ways:

1. Programs are extended with a component of signature $Id^c \times Id^f \rightarrow c$, which maps the name of a class and a field to the field's type; to look up the type of a field we use ${}^{f}P{\downarrow_4}(c, f)$.

2. Objects are extended to contain a mapping of field names to values. The definition of the heap is accordingly extended.

3. To cater for fields, two rules are included which read from and write to a field. The rule for object creation is also extended to initialise the field-to-value mapping of a new object with *null* values.

4. The type system is extended with two judgements for field reads and writes and a judgement for *null* values.

5. The definition of a well-formed class is extended to require that any field declared in a class is inherited and not overridden in a subclass (definition 6).

Abstract Syntax

$$
\begin{array}{rcl}
{}^{f}e \in Expr & ::= & null \mid this \mid x \mid new\ c \mid {}^{f}e.m({}^{f}e) \mid {}^{f}e; {}^{f}e \mid {}^{f}e.f \mid {}^{f}e.f = {}^{f}e \\
MethSig & ::= & t\ m(c) \\
t \in Type & ::= & void \mid c \\
x, c, m, f \in Id & &
\end{array}
$$

Program Representation

$$
\begin{array}{rcl}
Program & = & (Id^c \to Id^c) \times (Id^c \times Id^m \to MethSig) \times (Id^c \to \mathcal{P}(Chord)) \\
 & & \quad \times (Id^c \times Id^f \to c) \\
Chord & = & (Id^m \cup \{\varepsilon\}) \times \mathcal{P}(Id^m) \times Expr
\end{array}
$$

Runtime Entities

$$
\begin{array}{rcl}
Configuration & = & RExpr^\star \times Heap \\
Heap & = & \mathbb{N} \to Id^c \times Fields \\
Fields & = & Id^f \rightharpoonup \mathbb{N} \\
{}^{f}re \in RExpr & ::= & voidValue \mid null \mid \iota \mid new\ c \mid {}^{f}re.m({}^{f}re) \mid {}^{f}re; {}^{f}re \\
 & \mid & {}^{f}re.f \mid {}^{f}re.f = {}^{f}re \\
\iota \in \mathbb{N} & &
\end{array}
$$

Figure 3.7: ^{f}SCHOOL overview.

6. A new definition for well-formed heaps is introduced to ensure that the contents of fields are as according to their static types (definition 8).

The program of listing 3.3 is represented as in figure 3.8; it is similar to the original representation from figure 3.2 of section 3.3.1, but now includes a fourth element for the field `permits`.

3.4.2 Types, Well-Formedness and Subject Reduction

We extend the notions of well-formedness to ^{f}SCHOOL. Since class and program well-formedness in ^{f}SCHOOL *replaces* the requirements from SCHOOL,

$$
\begin{array}{l}
{}^{f}P{\downarrow_1}(CountdownLatch) = Object \\
{}^{f}P{\downarrow_2}(CountdownLatch, await) = void\ await(int) \\
{}^{f}P{\downarrow_2}(CountdownLatch, countDown) = void\ countDown(Object) \\
{}^{f}P{\downarrow_3}(CountdownLatch) = \{(await, \{countDown\}, this.permits = this.permits - 1; \\
\quad if(this.permits > 0)\ this.await(null);)\} \\
{}^{f}P{\downarrow_4}(CountdownLatch, permits) = int
\end{array}
$$

Figure 3.8: ^{f}SCHOOL representation of the Countdown Latch program.

we use the symbol $\vdash_{\mathcal{f}}$ to disambiguate. We do not need to do the same for execution (figure 3.9) and types (figure 3.10), because the ^{f}SCHOOL evaluation rules and typing judgments *extend* those of SCHOOL.

An ^{f}SCHOOL class c is well-formed, denoted ${}^{f}P \vdash_{\mathcal{f}} c$, if it well-formed in SCHOOL, and another two additional requirements are met: first, if a field has as a type a class c', then c' must be declared; second, the type of a field in a class must match the type of the field in the superclass:

Definition 6 (^{f}SCHOOL **Additional Well-Formed Classes)**
${}^{f}P \vdash_{\mathcal{f}} c$ iff :

1. $P \vdash c$
2. ${}^{f}P{\downarrow_4}(c, f) = c' \implies {}^{f}P \vdash_{\diamond_{cl}} c'$
3. ${}^{f}P{\downarrow_4}({}^{f}P{\downarrow_1}(c), f) = t \implies {}^{f}P{\downarrow_4}(c, f) = t$

As in SCHOOL, an ^{f}SCHOOL program is well-formed, denoted $\vdash_{\mathcal{f}} {}^{f}P$, if all declared classes are well-formed:

Definition 7 (^{f}SCHOOL **Well-Formed Programs)**
$\vdash_{\mathcal{f}} {}^{f}P$ iff ${}^{f}P \vdash_{\diamond_{cl}} c \implies {}^{f}P \vdash_{\mathcal{f}} c$

An ^{f}SCHOOL heap ${}^{f}h$ is well-formed, denoted ${}^{f}P \vdash_{\mathcal{f}} {}^{f}h$, if each field of each object can be typed:

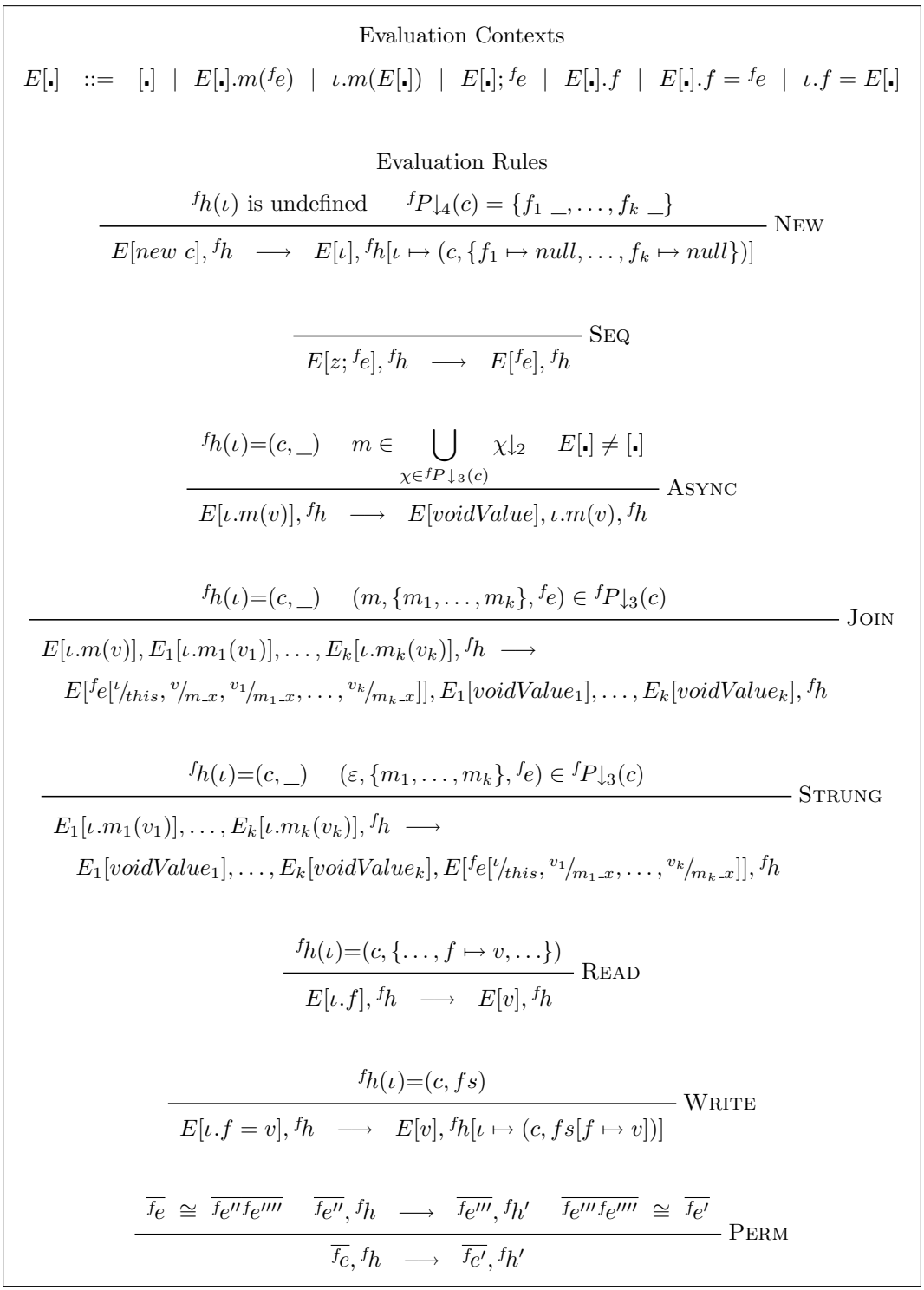

Figure 3.9: ^{f}SCHOOL operational semantics.

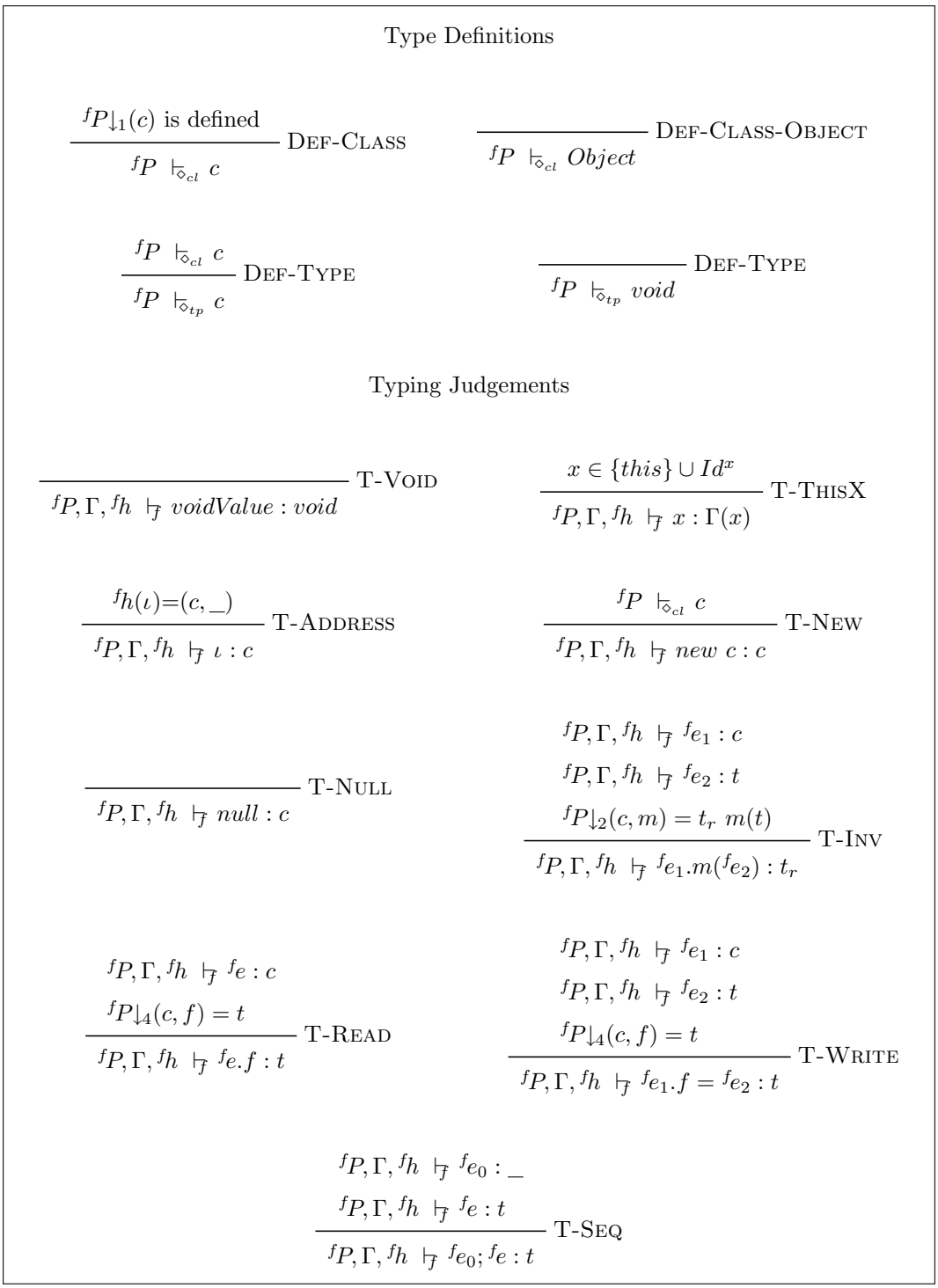

Figure 3.10: ^{f}SCHOOL type system.

Definition 8 (^{f}SCHOOL Well-Formed Heaps)

$$
{}^{f}P \ \vdash_{\mathcal{f}} {}^{f}h \ \text{ iff } \ \forall\, \iota \in {}^{f}h \ : \\
\left.\begin{array}{l} {}^{f}h(\iota){=}(c, fs) \\ P \ \vdash_{\Diamond_{cl}} c \\ {}^{f}P{\downarrow_4}(c, f) = c' \end{array}\right\} \implies {}^{f}P, \_, {}^{f}h \ \vdash_{\mathcal{f}} fs(f) : c'
$$

We can show that ^{f}SCHOOL is sound by proving subject reduction; the theorem and proof are similar to those for SCHOOL. Because ^{f}SCHOOL expressions may contain field accesses, and a field's value may be an address, an expression may evaluate to an address not initially present. Therefore, we need to also take care of the objects' field values, and hence we also require ${}^{f}P \ \vdash_{\mathcal{f}} {}^{f}h$.

Theorem 3 (^{f}SCHOOL **Subject Reduction)**

$$
\left.\begin{array}{l} \vdash_{\mathcal{f}} {}^{f}P \\ {}^{f}P \ \vdash_{\mathcal{f}} {}^{f}h \\ {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\mathcal{f}} \overline{fe} : \bar{t} \\ \overline{fe}, {}^{f}h \ \longrightarrow \ \overline{fe'}, {}^{f}h' \end{array}\right\} \implies \left\{\begin{array}{l} {}^{f}P \ \vdash_{\mathcal{f}} {}^{f}h' \ \ \wedge \\ \exists\, \overline{t'} \ \cong \ \bar{t}, t'' \text{ where } t'' = \mathit{void} \text{ or } t'' = \varepsilon \ : \\ \quad {}^{f}P, \Gamma, {}^{f}h' \ \vdash_{\mathcal{f}} \overline{fe'} : \overline{t'} \end{array}\right.
$$

Proof *We first prove a theorem for a single expression, similar to theorem 1 for SCHOOL, and then use that in combination with structural induction over the derivation of* $\overline{fe}, {}^{f}h \ \longrightarrow \ \overline{fe'}, {}^{f}h'$*; we perform a case analysis of the last evaluation rule applied.* $\square$

3.4.3 Progress

Progress for ^{f}SCHOOL can be demonstrated in the same manner as for SCHOOL. The two new evaluation rules, Read and Write, ensure that all field reading and writing expressions can evaluate, and hence never contribute towards blocking the configuration; furthermore, the resulting values from reading from and writing to a field will be discarded through the Seq rule in the event of sequential composition, which ensures progress in such a case.

```
1  class CountdownLatch {
2   CountdownLatch init_CountdownLatch() {
3    permits(0); return this;
4   }
5   int get_permits() & async permits(int r) {
6    permits(r); return r;
7   }
8   int set_permits(int s) & async permits(int r) {
9    permits(s); return s;
10  }
11  void await() & async countDown() {
12   set_permits(get_permits() - 1);
13   if (get_permits() > 0) { await(); }
14  }
15 }
```

Listing 3.4: Encoded chorded Countdown Latch with fields.

Theorem 4 (^{f}SCHOOL Progress)

$$\left.\begin{array}{l} \vdash_{\mathcal{f}} {}^fP \\ {}^fP \vdash_{\mathcal{f}} {}^fh \\ {}^fP, \_, {}^fh \vdash_{\mathcal{f}} \overline{fe} : \overline{t} \end{array}\right\} \implies \left\{\begin{array}{ll} & \exists\, \overline{fe'}, {}^fh' \ : \ \overline{fe}, {}^fh \longrightarrow \overline{fe'}, {}^fh' \\ \vee & \overline{fe} \text{ consists of ground values or} \\ & \text{null-pointer expressions only} \\ \vee & \overline{fe}, {}^fh \text{ is blocked} \end{array}\right.$$

Proof *Similar to that of theorem 2 with two new straightforward cases for the evaluation rules* Read *and* Write. *An extra requirement in the case of* ^{f}SCHOOL *is that fields defined in a superclass must also be defined in a subclass, but this is guaranteed by well-formed programs.* □

3.5 Encoding ^{f}SCHOOL into SCHOOL

We obtain an encoding of ^{f}SCHOOL programs into SCHOOL by employing chords in order to record the current value of each field, read and write values

of fields, and to initialise each field with a default value upon object creation.

Listing 3.4 demonstrates the translation of the chorded countdown latch with fields. There are four chords in the translated class: the first chord, `init_CountdownLatch`, describes object initialization: we create a new countdown latch through `new CountdownLatch().init_CountdownLatch()`. The second chord, with synchronous part `get_permits`, describes reading of field `permits`; through `o.get_permits()` we read the field `permits` from the object `o`. The third chord, with synchronous part `set_permits(int r)`, describes writing field `permits`; through `o.set_permits( v )` we set the field `permits` of the object `o` to the value `v`. The fourth chord, with synchronous part `await`, corresponds to that from listing 3.3, where field reads/writes have been replaced by calls to `get_permits` and `set_permits`. Furthermore, in listing 3.4 there is the asynchronous method `permits` whose parameter represents the value of the field permits.

Using `new CountdownLatch.init_CountdownLatch()` we create a new countdown latch, and before the new object is returned the invocation of `permits( 0 )` will be available in the execution.

Figure 3.11 presents the definition of encoding $\mathbb{C}$, from ^{f}SCHOOL to SCHOOL, for expressions and for programs, where chords encode the behaviours of fields (mapping to values, reading, and writing).

For a class c, with a field f of type t, the encoding creates the following method signatures:

$$\begin{aligned} P{\downarrow_2}(c, f) &= void\ f(t) \\ P{\downarrow_2}(c, get\_f) &= t\ get\_f(Object) \\ P{\downarrow_2}(c, set\_f) &= t\ set\_f(t) \\ P{\downarrow_2}(c, init\_c) &= c\ init\_c(Object) \end{aligned}$$

The argument type of f is t, to represent the type of the field, and the result type is *void*, since f is used in the asynchronous parts of the chords. The argument of get_f is discarded, and its type is *Object* only in order to

Encoding Expressions

$$\mathbb{C}({}^f e) \;=\; {}^f e \text{ if } {}^f e \in \{voidValue, \iota, null, this, x\}$$

$$\mathbb{C}(new\ c) \;=\; new\ c.init\_c(null) \qquad \mathbb{C}({}^f e.m({}^f e')) \;=\; \mathbb{C}({}^f e).m(\mathbb{C}({}^f e'))$$

$$\mathbb{C}({}^f e.f) \;=\; \mathbb{C}({}^f e).get\_f(null) \qquad \mathbb{C}({}^f e.f = {}^f e') \;=\; \mathbb{C}({}^f e).set\_f(\mathbb{C}({}^f e'))$$

Encoding Programs

$\mathbb{C}({}^f P) = P$ iff :

1. ${}^f P{\downarrow_1}(c) = P{\downarrow_1}(c)$

2. $$t\ m(t') = P{\downarrow_2}(c,m) \iff \begin{cases} t = void,\ m = init\_c,\ t' = Object \\ \vee \quad f\ t \in {}^f P{\downarrow_4}(c) \ \wedge \begin{cases} t' = Object,\ m = get\_f \\ \vee \quad t' = t,\ m = set\_f \\ \vee \quad t = void,\ m = f \end{cases} \\ \vee \quad t\ m(t') \in {}^f P{\downarrow_2}(c,m) \end{cases}$$

3. $$(m, \{m_1, \ldots, m_k\}, {}^f e) \in {}^f P{\downarrow_3}(c) \implies (m, \{m_1, \ldots, m_k\}, \mathbb{C}({}^f e)) \in P{\downarrow_3}(c)$$

4. $$f\ \_ \in {}^f P{\downarrow_4}(c) \implies \begin{cases} (get\_f, \{f\}, this.f(f\_x); f\_x;) \in P{\downarrow_3}(c) \\ \wedge \quad (set\_f, \{f\}, this.f(set\_f\_x); set\_f\_x;) \in P{\downarrow_3}(c) \end{cases}$$

5. $$\{f_1\ \_, \ldots, f_k\ \_\} \in {}^f P{\downarrow_4}(c) \iff (init\_c, \emptyset, this.f_1(null); \ldots; this.f_k(null); this;) \in P{\downarrow_3}(c)$$

6. $${}^f P \vdash_{\diamond_{cl}} c \iff (init\_c, \emptyset, \_) \in P{\downarrow_3}(c)$$

Figure 3.11: Encoding ^{f}SCHOOL into SCHOOL.

satisfy our restriction to single-parameter methods. The result type is t. The argument and the result type of set_f is t, to represent the fact that the field

has type t, and that the new field value is being returned. The argument type of $init\_c$ is $Object$ again for the same reason as it was for get_f, and the result type is t, because the newly initialized object is returned.

The field-reading chord then has the following form:

$$(get\_f, \{f\}, this.f(f\_x); f\_x) \in P{\downarrow_3}(c)$$

The joining of this chord results in the f invocation being consumed, its argument, f_x, eventually being returned as the value of the field. However, as fields should not be consumed by reading, the invocation of f must first be placed back into the execution, via $this.\, f(f\_x)$, so that further reads and writes can take place.

The field writing chord has the following form:

$$(set\_f, \{f\}, this.f(set\_f\_x); set\_f\_x) \in P{\downarrow_3}(c)$$

Upon joining, the argument to the set_f method is placed back into the execution (as well as returned). The old value of the field (the argument to the f method) is lost.

From the above two chords we see that the invocation to f acts as a "mutex" to the encoded field; at any given point of execution only one chord instance can have access to the invocation. Only after the chord places the invocation back into the execution can another chord join and consume the invocation. Hence, field reading and writing are atomic, which corresponds with the the atomic evaluation of the ^{f}SCHOOL rules Read and Write.

Finally, assuming that the class has fields $f_1, \ldots, f_n$, the chord to initialise the fields is the following:

$$(init\_c, \emptyset, this.f_1(null); \ldots; this.f_n(null); this) \in P{\downarrow_3}(c)$$

This chord will join when $init\_c$ is invoked, and place an $\iota.\, f_i(null)$ invocation into the execution for each field f_i of the object being initialised, with

the default value as argument. Finally, it returns the current object, ι, to its invoker.

We assume that none of the methods in the ^{f}SCHOOL program has identifier $get\_\ldots$ or $set\_\ldots$ or $init\_\ldots$, nor the name of a field f from that program.

The return type of the $init\_\ldots$ method is dependent on the class name c, as the return type will be different for a subclass of c; thus, in order for the encoded program to type check, we require a unique name for the initialisation chord's method (for instance, class D which extends class C will result in two return types for $init\_\ldots$, D and C in each class respectively). Were the type system to support covariant overriding of methods, we could use a single name *init* for all initialisation chords.

Encoding of fields using other language constructs has been studied by amongst others Abadi and Cardelli [1] which uses method invocations and updates in order to obtain the behaviour of fields; this encoding is particularly suitable for functional calculi, as invoking methods and updating their bodies for the remainder of the evaluation does not require assignment to previously defined variables. ^{f}SCHOOL, however, focuses on the mutable field construct of many chorded languages and thus does not benefit from a purely functional basis; the encoding of fields for SCHOOL presented above is therefore fundamentally different. Furthermore, the use of asynchronous invocations to represent the current values of fields gives us the framework below for showing correspondence which is not available in the Abadi and Cardelli encoding, as their calculus has no provision for concurrency.

3.5.1 Properties of the Encoding

The goal of the encoding is to show that the behaviour of fields can be obtained using only chords. This essentially means that the encoding satisfied a notion of *soundness* and a notion of *completeness*, where soundness ensures that the behaviours of the original programs (which feature chords and fields) are pre-

served in the encoded programs (which feature only chords), and completeness ensures that encoded programs do not introduce new behaviours which do not correspond to those of the original programs.

Hence, configurations of original and translated programs must correspond, and execution of both original and encoded programs must lead to respectively corresponding configurations. In the next section we describe *strong correspondence* between an ^{f}SCHOOL configuration and its encoding in SCHOOL when expressions, object classes, and field current values are preserved, denoted $\overline{{}^{f}e}, {}^{f}h \simeq \overline{e}, h$, with which we show soundness of the encoding.

However, showing completeness is not as straightforward because certain steps of evaluation in ^{f}SCHOOL programs require several *intermediate* steps of evaluation, as well as additional concurrent expressions in the encoded SCHOOL program. Field-access chords must join with, and restore, the asynchronous method invocation representing the field's current value, and object-initialisation chords must populate the configuration with asynchronous invocations consisting of initial field values. Furthermore, due to concurrency, these multiple steps and additional expressions can also be arbitrarily interleaved. Thus, new behaviours can be generated which are not the direct result of the encoding, rather, they depend on non-deterministic interleaving.

Consequently, strong correspondence does not suffice to show completeness, since we need to ensure that although intermediate steps of evaluation "temporarily" break correspondence (such as the current values of fields which are missing immediately after the joining of a field-access chord), eventually, original and encoded programs always further execute and reach strongly corresponding configurations again. Hence, we introduce *weak correspondence*, denoted $\overline{{}^{f}e}, {}^{f}h \approx \overline{e}, h$, which allows for these temporary discrepancies during evaluation of intermediate steps.

$$\frac{\forall\, \iota \in dom({}^{f}h)\ :\ {}^{f}h(\iota){=}(c,\_)\ \Longrightarrow\ h(\iota){=}c \qquad \overline{e} = \{\iota.f(v) : {}^{f}h(\iota)\downarrow_2(f) = v\}}{{}^{f}e_1,\ldots,{}^{f}e_n,{}^{f}h\ \simeq\ \mathbb{C}({}^{f}e_1),\ldots,\mathbb{C}({}^{f}e_n),\overline{e},h}\ \text{C-Strong}$$

$$\frac{\overline{{}^{f}e}\ \cong\ \overline{{}^{f}e'} \qquad \overline{{}^{f}e'},{}^{f}h\ \simeq\ \overline{e'},h \qquad \overline{\mathit{voidValue}}\ \overline{e'}\ \cong\ \overline{e}}{\overline{{}^{f}e},{}^{f}h\ \simeq\ \overline{e},h}\ \text{C-Strong-Perm}$$

Figure 3.12: Strong Correspondence

3.5.2 Strong Correspondence

In figure 3.12 we define strong correspondence between ^{f}SCHOOL and SCHOOL configurations, in the form of the judgement $\overline{{}^{f}e},{}^{f}h\ \simeq\ \overline{e},h$.

The first rule, C-Strong, ensures that an ^{f}SCHOOL configuration corresponds with its encoded SCHOOL configuration since it reproduces the behaviours of ^{f}SCHOOL expressions and preserves the classes of objects and the current values of fields, by requiring the following:

1. For each expression ${}^{f}e_i$ in the ^{f}SCHOOL configuration, its encoding $\mathbb{C}({}^{f}e_i)$ appears as an expression in the SCHOOL configuration.

2. All objects in the ^{f}SCHOOL heap ${}^{f}h$ are also present in the SCHOOL heap h and have the same class.

3. The SCHOOL configuration contains expressions representing the contents of the fields in the ^{f}SCHOOL heap ${}^{f}h$.

Note that in the third requirement we use the sequence $\overline{e}$ as a set, when we require that $\overline{e} = \{\ldots\}$, and we express that the sequence should contain exactly the elements in this set.

The second rule, C-Strong-Perm, enables the application of the first rule onto any permutation of the expressions, and allows for any number of additional *voidValue* ground expressions in the SCHOOL configuration (left over from the joining of field-access chords).

Consider as an example the encoding of a program which employes a countdown latch (at address ι_0) where the current value of the `permits` field is 3; the ^{f}SCHOOL heap is the following:

$$ {}^f h = \{\iota_0 \mapsto (CountdownLatch, \{permits \mapsto 3\})\} $$

and the current configuration is:

$$ \iota_0.permits = 2, {}^f h $$

The encoded heap in SCHOOL is:

$$ h = \{\iota_0 \mapsto CountdownLatch\} $$

and the encoded configuration in SCHOOL is:

$$ \iota_0.set\_permits(2), \iota_0.permits(3), h $$

By C-Strong the original program and its encoding indeed correspond:

$$ \iota_0.permits = 2, {}^f h \ \simeq \ \iota_0.set\_permits(2), \iota_0.permits(3), h $$

If, however, there were two invocations of the `permits` method present in the SCHOOL configuration, the correspondence would not hold:

$$ \iota_0.permits = 2, {}^f h \ \not\simeq \ \iota_0.set\_permits(2), \iota_0.permits(3), \iota_0.permits(4), h $$

Execution of the ^{f}SCHOOL program results in a configuration where 2 becomes the current expression, as well as the current value of the `permits` field (through the application of the Write rule):

$$ \iota_0.permits = 2, {}^f h \ \xrightarrow{\text{Write}} \ 2, {}^f h' $$
$$ \text{where} \quad {}^f h' = {}^f h[\iota_0 \mapsto (CountdownLatch, \{permits \mapsto 2\})] $$

The corresponding initial SCHOOL configuration reaches the corresponding final configuration as shown in figure 3.13.

Using strong correspondence we can show soundness (theorem 5 below); i.e. if the ^{f}SCHOOL configuration makes a step, then it is *always possible*

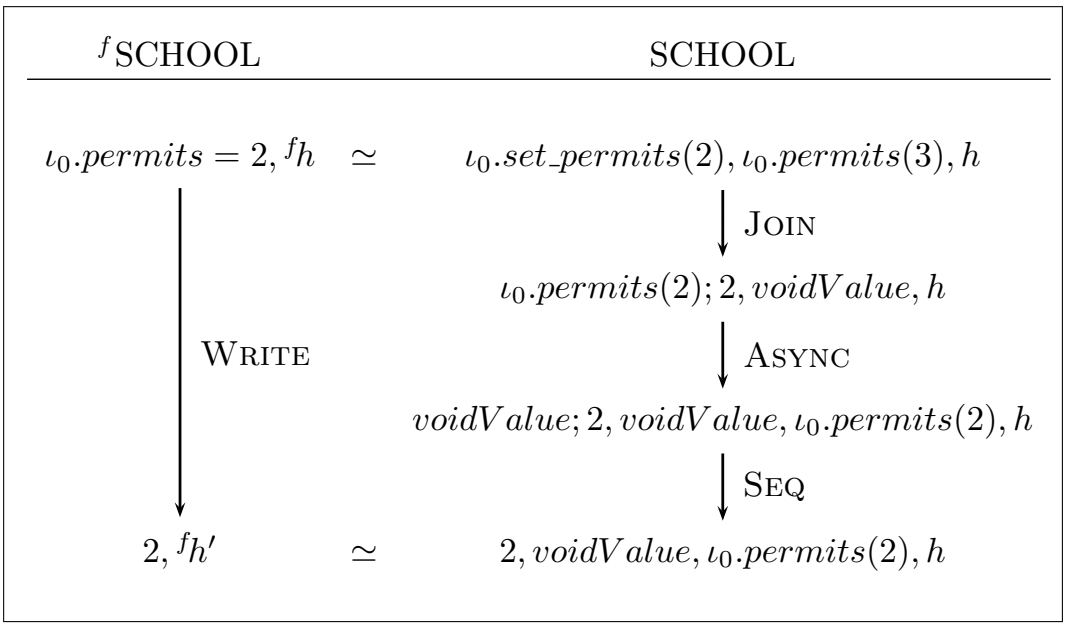

Figure 3.13: Example of strong correspondence.

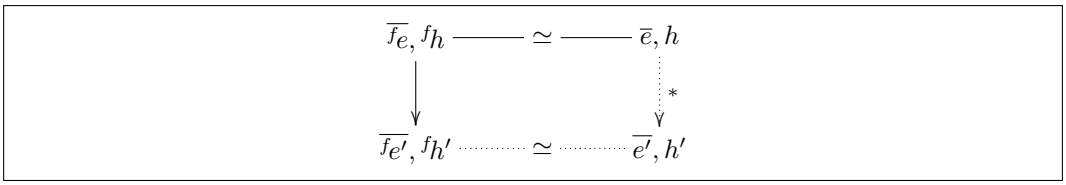

Figure 3.14: Soundness of the encoding.

to evaluate the strongly corresponding SCHOOL configuration, possibly in multiple steps, so that the final configurations also correspond strongly.

Note that in the theorem below we assume the program ^fP to be global, in the sense that it is the same program which executes in ^{f}SCHOOL on both sides of the implication, and additionally the SCHOOL execution is that of the program $\mathbb{C}(^fP)$.

Theorem 5 (Soundness of the Encoding)

$$\left.\begin{array}{l} \overline{{}^fe}, {}^fh \simeq \overline{e}, h \\ \overline{{}^fe}, {}^fh \longrightarrow \overline{{}^fe'}, {}^fh' \end{array}\right\} \implies \exists\, \overline{e'}, h' \,:\, \overline{e}, h \longrightarrow^* \overline{e'}, h' \quad \wedge \quad \overline{{}^fe'}, {}^fh' \simeq \overline{e'}, h'$$

Proof *We first prove a lemma for a single thread by case induction on the execution* $^fe, {}^fh \longrightarrow \overline{{}^fe'}, {}^fh'$*; the theorem for many threads then follows.* □

The diagram from figure 3.14 illustrates the above theorem; solid lines represent the premises and dotted lines the conclusions.

3.5.3 Weak Correspondence

In the example from figure 3.13 we notice that between strongly corresponding configurations, intermediate configurations occur. The aim of weak correspondence is to ensure that although strong correspondence (e.g. field current values) is not preserved during intermediate steps, eventually, we reach configurations which correspond strongly again. Hence, using strong and weak correspondence together, original and encoded program executions correspond throughout.

Under weak correspondence, two field-access chords will never *simultaneously* join for the *same* field of the *same* object. Furthermore, no field-access chord will join before an initial value for the field has been provided by the initialisation chord's body.

Weak correspondence keeps track of which objects and fields are currently being accessed by the SCHOOL expressions of the encoding, and does not allow any other expressions to access them simultaneously. Essentially, the "mutex" behaviour of asynchronous methods representing field current values is enforced.

In figures 3.15 and 3.17 we define two forms of the weak correspondence judgement. The first has the form $\varphi \vdash {}^{f}e, {}^{f}h \approx \overline{e}, h$ and is between an individual ^{f}SCHOOL expression and a heap and potentially multiple SCHOOL expressions and a heap. The second has the form $\overline{{}^{f}e}, {}^{f}h \approx \overline{e}, h$ and is between complete ^{f}SCHOOL and SCHOOL configurations, which allows us to apply the individual expression form for each expression in the ^{f}SCHOOL configuration. Additionally, we define heap correspondence, which has the form ${}^{f}h \approx h$, and which ensures that all objects in an ^{f}SCHOOL heap also exist in the SCHOOL heap and have the same class in both heaps.

$$\frac{\varphi \vdash {}^f e, {}^f h \;\approx\; e, \overline{e}, h}{\varphi \vdash {}^f e, {}^f h \;\approx\; z; e, \overline{e}, h}\ \text{C-Wk-Void} \qquad \frac{}{\varepsilon \vdash v, {}^f h \;\approx\; v, h}\ \text{C-Wk-Val}$$

$$\frac{\varphi \vdash {}^f e, {}^f h \;\approx\; e, \overline{e}, h}{\varphi \vdash {}^f e.f, {}^f h \;\approx\; e.get\_f(null), \overline{e}, h}\ \text{C-Wk-Read}$$

$$\frac{{}^f h(\iota) = (\_, \{\ldots, f \mapsto v, \ldots\})}{\iota.f \vdash \iota.f, {}^f h \;\approx\; \iota.f(v); v, h}\ \text{C-Wk-Read-Body}$$

$$\frac{\varphi \vdash {}^f e, {}^f h \;\approx\; e, \overline{e}, h \qquad \varphi' \vdash {}^f e', {}^f h \;\approx\; e', \overline{e'}, h \qquad {}^f e = v \text{ or } \varphi' = \varepsilon}{\varphi \oplus \varphi' \vdash {}^f e.f = {}^f e', {}^f h \;\approx\; e.set\_f(e'), \overline{e}, \overline{e'}, h}\ \text{C-Wk-Write}$$

$$\frac{{}^f h(\iota) \text{ is defined}}{\iota.f \vdash \iota.f = v, {}^f h \;\approx\; \iota.f(v); v, h}\ \text{C-Wk-Write-Body}$$

$$\frac{}{\varepsilon \vdash new\ c, {}^f h \;\approx\; new\ c.init\_c(null), h}\ \text{C-Wk-New}$$

$$\frac{{}^f h(\iota) \text{ is undefined} \quad h(\iota) = c}{\iota \vdash new\ c, {}^f h \;\approx\; \iota.init\_c(null), h}\ \text{C-Wk-New-Begin}$$

$$\frac{h(\iota) = c \quad {}^f h(\iota) \text{ is undefined} \quad {}^f P{\downarrow_4}(c) = \{f_1\ \_, \ldots, f_k\ \_\} \quad r < k}{\begin{array}{c}\iota \vdash new\ c, {}^f h \;\approx\; \iota.f_r(null); \ldots; \iota.f_k(null); \iota, \\ \iota.f_1(null), \ldots, \iota.f_{r-1}(null), h\end{array}}\ \text{C-Wk-New-Body}$$

$$\frac{\varphi \vdash {}^f e, {}^f h \;\approx\; e, \overline{e}, h \qquad \varphi' \vdash {}^f e', {}^f h \;\approx\; e', \overline{e'}, h \qquad {}^f e = v \text{ or } \varphi' = \varepsilon}{\varphi \oplus \varphi' \vdash {}^f e.m({}^f e'), {}^f h \;\approx\; e.m(e'), \overline{e}, \overline{e'}, h}\ \text{C-Wk-Meth}$$

Figure 3.15: Weak Correspondence for Expressions

Individual Expressions

The form of weak correspondence for individual expressions (figure 3.15) is:

$$\varphi \vdash {}^f e, {}^f h \;\approx\; \overline{e}, h$$

where the ^{f}SCHOOL expression $^f e$ corresponds with potentially multiple SCHOOL expressions $\overline{e}$, and where an object or field indicated by φ is currently being accessed by the SCHOOL expressions.

When a field f of an object at address ι is being accessed (either by a field-read or field-write chord) the form of φ is $\iota.f$; when an entire object at address ι is being initialised by the *init_...* chord the form of φ is ι; when no field or object is being accessed then φ is empty, denoted ε. Hence, $\varphi \in Addr \times Id^f \cup Addr \cup \{\varepsilon\}$.

In figure 3.16 we illustrate the use of weak correspondence on the original example from figure 3.13, where an ^{f}SCHOOL expression writes the value 2 to the permits field. It is now possible to apply a weak correspondence judgement after each step of the SCHOOL execution, including intermediate configurations where the ^{f}SCHOOL configuration has not executed (denoted by a dotted line).

The initial configurations weakly correspond through the C-Wk-Write rule, whose premises are satisfied through the C-Wk-Val rule, applied both on the receiver address ι_0 and on the value 2, which appear in both configurations.

After the initial SCHOOL configuration evaluates through the Join rule (the ^{f}SCHOOL configuration does not evaluate), the invocation $\iota_0.permits(3)$ is missing from the configuration (it has been replaced by *voidValue*). Accordingly, φ contains $\iota_0.permits$ and allows us to judge the second SCHOOL configuration as weakly corresponding to the initial ^{f}SCHOOL configuration, through the C-Wk-Write-Body rule.

The next step of evaluation in SCHOOL is performed via the Async rule, and this time the ^{f}SCHOOL configuration makes its single step of evaluation (via the Write rule). The invocation representing the new current value of the *permits* field is once again present in the configuration, and accordingly φ becomes empty. Through the C-Wk-Void rule the two new configurations weakly correspond; the premise of the rule is satisfied through the C-Wk-Val

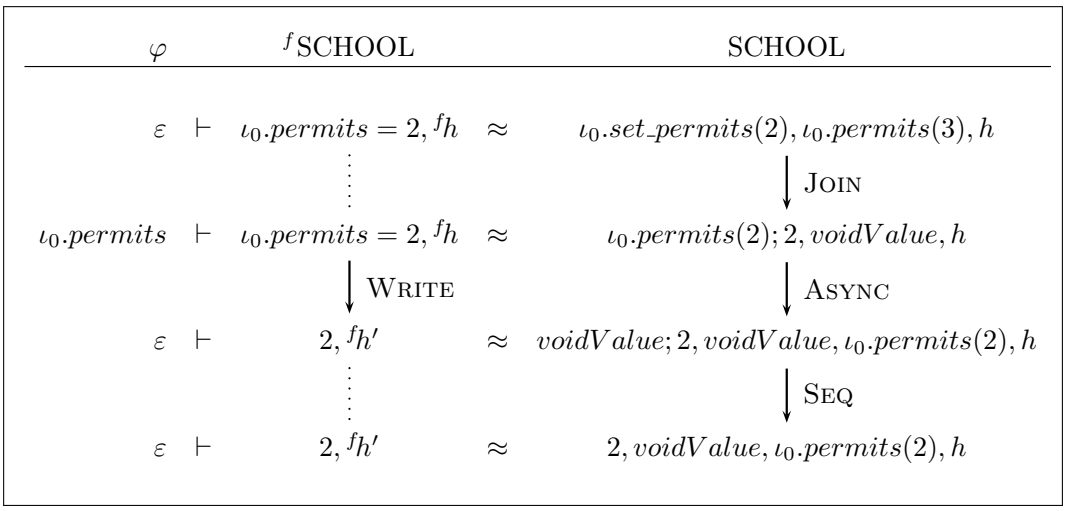

Figure 3.16: Example of weak correspondence.

rule, applied on the value 2.

The SCHOOL configuration makes a third, and final, step of evaluation (via the Seq rule), while the ^{f}SCHOOL configuration performs no further evaluation; the two final configurations weakly correspond via the C-Wk-Val rule (and hence φ is empty).

When an expression contains sub-expressions (such as the receiver and argument of a method invocation) we combine φ and φ' through the $\oplus$ operation, defined as follows:

$$\varphi \oplus \varphi' = \begin{cases} \varphi & \text{if } \varphi' = \varepsilon \\ \varphi' & \text{if } \varphi = \varepsilon \\ \text{undefined} & \text{otherwise} \end{cases}$$

Notice that the above definition, by requiring that one of the φs is empty, prohibits sub-expressions from simultaneously accessing fields, even when those fields are distinct.

This restriction on $\oplus$ is desirable because if we allowed sub-expressions to simultaneously access fields we would consider as corresponding SCHOOL configurations which are unreachable. For example, the SCHOOL configuration $E[\iota.f(v); v].m(\iota.f'(v'); v'), h$ is not reachable due to the evaluation order

imposed by contexts.

The following weak correspondence rules are between a single ^{f}SCHOOL expression and its encoding:

- C-Wk-Void enables us to dispense with the sequential composition of *voidValue* with any other expression.

- C-Wk-Val indicates that ground values immediately correspond, and $\varphi = \varepsilon$ since the SCHOOL expression is either not in a field-access chord body, or is the last term of a field-access chord body and hence has already made the asynchronous invocation representing the field's current value.

- C-Wk-Read preserves the weak correspondence for the receiver object of a field-read method call; since the field-read chord has not yet joined the φ remains unchanged.

- C-Wk-Read-Body says that the ^{f}SCHOOL term $\iota.\, f$ corresponds to the sequence of SCHOOL expressions comprising the body of the field-reading chord, and therefore $\varphi = \iota.\, f$. Since reading a field does not change its value, the rule requires that the value passed as an argument to f, as well as the value returned, must be the value currently mapped to the field in ^{f}h.

- C-Wk-Write is similar to the C-Wk-Read rule; weak correspondence must be preserved for both the receiver and argument, and hence we use the combination $\varphi \oplus \varphi'$. The third premise of this judgement prohibits the receiver and argument from simultaneously accessing fields (as this would allow unreachable states due to the order imposed by contexts) by requiring either the receiver to be a ground value or the argument to not be accessing a field.

- C-Wk-Write-Body is similar to the C-Wk-Read-Body rule; however, since the current value of a field is lost during writing, the value mapped to the field in the ^{f}SCHOOL heap is ignored.

- C-Wk-New tells us that the object creation term in ^{f}SCHOOL directly corresponds with its encoding, and $\varphi = \varepsilon$ since the object initialisation chord has not joined yet.

- C-Wk-New-Body says that the ^{f}SCHOOL term for object creation corresponds with a sequence of SCHOOL expressions which consist of the partially evaluated body of the $init\_\ldots$ chord, and the asynchronous invocations to the f methods which have already been evaluated, and spawned through Async. Notice that this is the only case where an ^{f}SCHOOL expression corresponds to more than one SCHOOL expression.

- C-Wk-Meth tells us that a method call in ^{f}SCHOOL directly corresponds with its encoding in SCHOOL, as long as the receivers and the arguments also correspond, and hence we use the combination $\varphi \oplus \varphi'$. Similar to C-Wk-Write above, the receiver and argument are prohibited from simultaneously accessing fields.

Configurations

The form of weak correspondence for configurations (figure 3.17) is:

$$\overline{{}^f e}, {}^f h \;\approx\; \overline{e}, {}^f h$$

and expresses that each of the expressions in $\overline{{}^f e}$ corresponds with exactly one sequence of expressions in $\overline{e}$, that no two SCHOOL expressions from $\overline{e}$ are bodies of field-access chords or object initialisation chords on the same field and object, and that all values of fields in $^f h$ are encoded by the presence of

$$\frac{{}^f h \approx h \quad \varphi_i \vdash {}^f e_i, {}^f h \;\approx\; \overline{e_i}, h \quad \overline{ef} = \mathcal{F}({}^f h, \varphi_1, \ldots, \varphi_n)}{{}^f e_1, \ldots, {}^f e_n, {}^f h \;\approx\; \overline{e_1}, \ldots, \overline{e_n}, \overline{ef}, h}\ \text{C-Wk}$$

$$\frac{\overline{{}^f e} \;\cong\; \overline{{}^f e'} \qquad \overline{{}^f e'}, {}^f h \;\approx\; \overline{e'}, h \qquad \overline{\mathit{voidValue}\ e'} \;\cong\; \overline{e}}{\overline{{}^f e}, {}^f h \;\approx\; \overline{e}, h}\ \text{C-Wk-Perm}$$

$$\frac{\forall\, \iota \in \mathit{dom}({}^f h) \;:\; {}^f h(\iota){=}(c, \_) \implies h(\iota){=}c}{{}^f h \approx h}\ \text{C-Wk-Heaps}$$

Figure 3.17: Weak Correspondence for Configurations

asynchronous method invocations in $\overline{e}$; the last two are ensured by the following definition.

Definition 9 (Field Encoding)

$$\mathcal{F}({}^f h, \varphi_1, \ldots, \varphi_n) = \begin{cases} \left\{ \begin{array}{l} \iota.f(v) \mid {}^f h(\iota)\!\downarrow_2(f) = v \\ \qquad \wedge\ \forall\ i \in 1..n \;:\; \iota.f \neq \varphi_i \end{array} \right\} & \text{if } i \neq j \implies \text{distinct } \varphi_i, \varphi_j \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Two φs are distinct when they indicate different fields and objects; however, when one φ indicates an entire object then the other φ must not indicate a field of that same object (nor the object itself, of course). Hence, we define distinct φ and φ' as: $\iota.\, f$ and $\iota'.\, f'$ are distinct iff $\iota \neq \iota'$ or $f \neq f'$; also ι and $\iota'.\, f'$ are distinct iff $\iota \neq \iota'$; finally $\iota.\, f$ and ι' are distinct iff $\iota \neq \iota'$.

Hence the above definition ensures that SCHOOL expressions do not "clash", in the sense that they are not the bodies for field-access chords for the same object and field, or initialisations for the same object. In order to prove soundness and completeness we use this definition to show that asynchronous methods encoding current values of fields act as mutexes and ensure that *get* and *set*

methods cannot access these values simulteneously; specifically, this definition shows us which mutexes are "open" at each step of evaluation by inspecting the currently recorded φs.

The following weak correspondence rules are between an ^{f}SCHOOL and an SCHOOL configuration:

- C-Wk imposes the following three requirements:

    1. Each object in $^f h$ exists and has the same class in h, as per the C-Wk-Heaps rule.
    2. Each ^{f}SCHOOL expression corresponds to exactly one sequence of SCHOOL expressions, and
    3. For each field f with value v in an object ι in $^f h$, either there exists a corresponding method invocation $\iota.f(v)$ (i.e., $\iota.f \in \varphi$ and $\overline{e}$ contains the appropriate invocation) or one of the SCHOOL expressions is currently executing a related access chord (i.e., $\iota.f = \varphi_i$, for some i, which implies $\varphi_i \vdash {}^f e_i, {}^f h \approx \overline{e_i}, h$).

- C-Wk-Perm extends the correspondence relationship to all possible permutations of the expressions, and allows any number of extra expressions containing *voidValue*.

Consider the example in figure 3.18 which features two Countdown Latch objects, at addresses ι_0 and ι_1, with current values 5 and 4, respectively. Weak correspondence allows for the simultaneous joining of two field-access chords on the same field of *two different objects* (and prohibits situations such as that of the example in figure 3.20, see later section 3.5.3). The first column indicates the values of φs when application of C-Wk rule gives us weak correspondence between the ^{f}SCHOOL and SCHOOL configurations in the second and third columns, respectively.

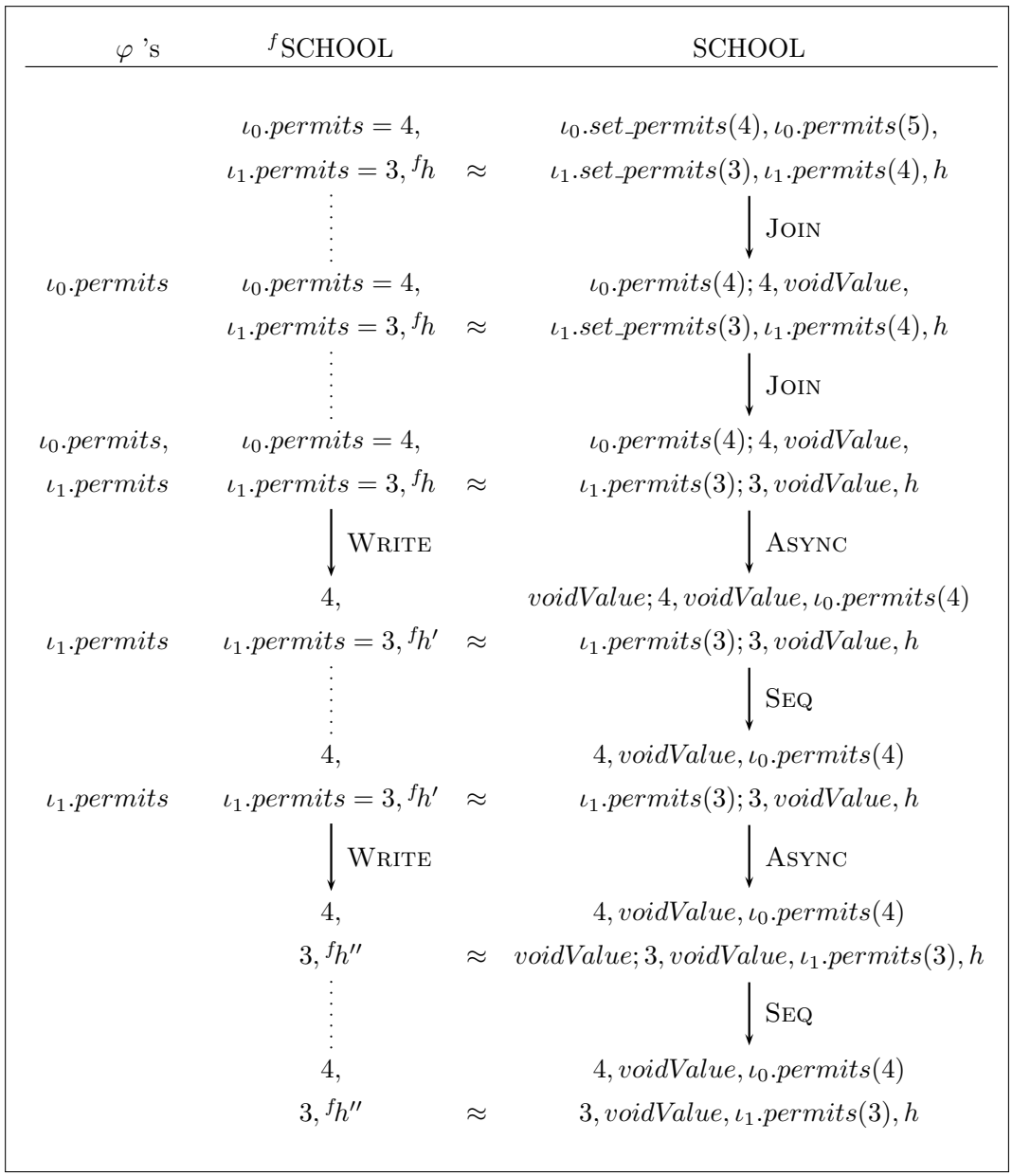

Figure 3.18: Example of weak correspondence with multiple fields.

We notice that after two steps of SCHOOL evaluation the φ of the top line is $\iota_0.permits$, while that of the bottom line is $\iota_1.permits$, and we can apply the C-Wk judgement. On the other hand, if the ^{f}SCHOOL configuration consisted of two concurrent writes to the *permits* fields of the *same* Countdown Latch, we would not be able to join both field-write chords in SCHOOL, as

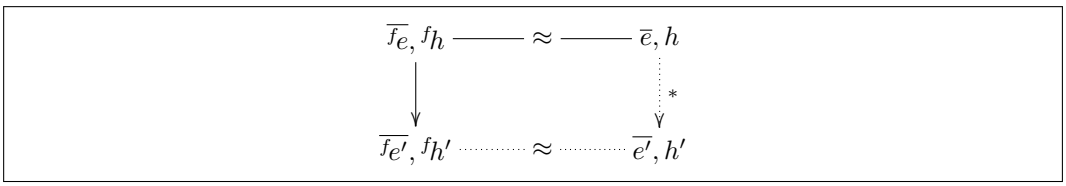

Figure 3.19: Soundness of the encoding with weak correspondence.

the φs would not be distinct, and hence the configurations would no longer weakly correspond.

Soundness

We show in lemma 4 that encoding preserves weak correspondence, and in theorem 6 we prove the stronger version of theorem 5.

Lemma 4 (Encoding Preserves Weak Correspondence)

(i) $\forall\ {}^{f}e, {}^{f}h, h\ :\ \varepsilon \vdash {}^{f}e, {}^{f}h\ \approx\ \mathbb{C}({}^{f}e), h$

(ii) $\forall\ \overline{{}^{f}e}, {}^{f}h, h\ :\ {}^{f}h \approx h\ \Longrightarrow\ \overline{{}^{f}e}, {}^{f}h\ \approx\ \overline{\mathbb{C}(e)}, h$

Proof *(i) by induction on the structure of* ${}^{f}e$*; (ii) by application of* C-Wk *and use of part (i) and the definition of* $\mathcal{F}$*.* $\square$

Theorem 6 (Soundness of The Encoding with Weak Correspondence)

$$\left.\begin{array}{l} \overline{{}^{f}e}, {}^{f}h\ \approx\ \overline{e}, h \\ \overline{{}^{f}e}, {}^{f}h\ \longrightarrow\ \overline{{}^{f}e'}, {}^{f}h' \end{array}\right\}\ \Longrightarrow\ \exists\ \overline{e'}, h'\ :\ \overline{e}, h\ \longrightarrow^{*}\ \overline{e'}, h'\ \wedge\ \overline{{}^{f}e'}, {}^{f}h'\ \approx\ \overline{e'}, h'$$

Proof *By structural induction on the derivation of* $\overline{{}^{f}e}, {}^{f}h\ \longrightarrow\ \overline{{}^{f}e'}, {}^{f}h'$ *and use of lemma 4; we perform a case analysis on the last evaluation rule applied.* $\square$

The diagram from figure 3.19 illustrates the above theorem; solid lines represent the premises of the theorem and dotted lines represent the conclusions.

Completeness

In order to show completeness we first show that individual ^{f}SCHOOL expressions and their corresponding sequences of SCHOOL expressions remain weakly corresponding during evaluation (lemma 5); the application of this result to multiple ^{f}SCHOOL expressions and their corresponding SCHOOL expressions constitutes completeness under weak correspondence (theorem 7). Finally, we show that strongly corresponding configurations can always further evaluate and reach strongly corresponding configurations (theorem 8).

From the two previous examples of figures 3.16 and 3.18 we notice that an intermediate SCHOOL configuration weakly corresponds to either the initial ^{f}SCHOOL configuration (when the ^{f}SCHOOL configuration has not executed, denoted by a dotted line), or weakly corresponds to the resulting ^{f}SCHOOL configuration after the ^{f}SCHOOL configuration has taken a single step of evaluation.

Indeed, this property forms the essence of weak correspondence, as we use it to show that all SCHOOL configurations (initial, intermediate or final) weakly correspond to some step of evaluation of the original ^{f}SCHOOL configuration (initial or final), and hence no new behaviours are introduced by the encoding.

An example where new behaviours are introduced in the absence of weak correspondence is in figure 3.20; here, two expressions assign values to the field *permits* of a Countdown Latch at address ι_0, which currently has the value 5. If its encoding in SCHOOL were to somehow allow both field-write chords to *simultaneously* join, then we would end up with a SCHOOL configuration where *two* invocations of $\iota_0.permits(\_)$ determine the current value of the *permits* field; such a configuration, and all of its further evaluations, would never lead to a strongly corresponding configuration.

When dealing with weak correspondence of an individual ^{f}SCHOOL expression we can ignore the corresponding sequence of asynchronous method

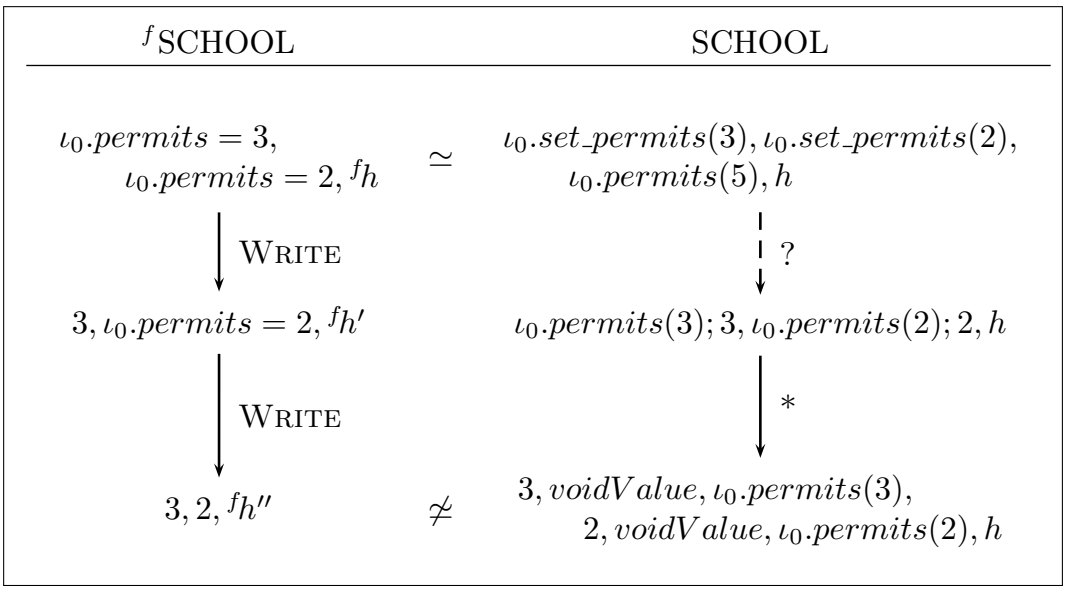

Figure 3.20: Example where strong correspondence is not reached.

invocations encoding field values, and hence write $\varphi \vdash {}^fe, {}^fh \;\approx\; e, \overline{e}, h$, where $e, \overline{e}$ is the corresponding sequence of SCHOOL expressions; however, when dealing with the execution of the corresponding SCHOOL configuration we do consider such invocations, and so write $e, \overline{e}, \overline{ef}, h \;\longrightarrow\; \overline{e'}, h'$, where $\overline{ef}$ is the sequence of these invocations.

Definition 10 below captures the relationship between an ^{f}SCHOOL step of evaluation and the current field access, determined by the value of φ. Because in ^{f}SCHOOL reading or writing a field is a single step, it is necessary that such a step does not occur *during* the access of a field in the corresponding SCHOOL configuration, as the field-access chord will not yet have replaced the asynchronous invocation encoding the field's current value, and correspondence would break. Hence, such ^{f}SCHOOL steps can occur only when φ' is empty.

Definition 10 (^{f}SCHOOL Step)

$$\varphi, {}^fe, {}^fh \sim \varphi', {}^fe', {}^fh' \text{ iff } \begin{cases} {}^fe = {}^fe' \wedge {}^fh = {}^fh' \\ \qquad \vee \\ {}^fe, {}^fh \;\longrightarrow\; {}^fe', {}^fh' \quad \wedge \quad \varphi' = \varepsilon \end{cases}$$

Using the above definition we show in lemma 5 below that when the SCHOOL configuration weakly corresponding to an individual ^{f}SCHOOL expression makes a single step of evaluation, the resulting configuration weakly corresponds to either the original ^{f}SCHOOL configuration or to the ^{f}SCHOOL configuration resulting from a step of evaluation. Furthermore, the sequence of asynchronous method invocations encoding field current values adheres to definition 9, taking into account when field-access chords return such an invocation and when they join and consume such an invocation.

Lemma 5 (Preservation of Weak Correspondence)

$$\left.\begin{array}{l} \varphi \vdash {}^f e, {}^f h \;\approx\; e, \overline{e}, h \\ \overline{ef} = \mathcal{F}({}^f h, \varphi, \varphi_1, \ldots, \varphi_k) \\ {}^f h \approx h \\ e, \overline{e}, \overline{ef}, h \;\longrightarrow\; e', \overline{e'}, \overline{ef'}, h' \end{array}\right\} \implies \begin{array}{l} \exists\;\; \varphi', {}^f e', {}^f h' \;: \\ \quad \varphi, {}^f e, {}^f h \sim \varphi', {}^f e', {}^f h' \\ \quad \varphi' \vdash {}^f e', {}^f h' \;\approx\; e', \overline{e'}, h' \\ \quad \overline{ef'} = \mathcal{F}({}^f h', \varphi', \varphi_1, \ldots, \varphi_k) \\ \quad {}^f h' \approx h' \end{array}$$

Proof *By structural induction on the judgement* $\varphi \vdash {}^f e, {}^f h \;\approx\; e, \overline{e}, h$ *and use of definition 10; we perform a case analysis on the weak correspondence judgement applied.*

The case for C-Wk-Val *is trivial as no evaluation rule is applicable; the case for* C-Wk-Void *is straightforward through the application of the inductive hypothesis; the case for* C-Wk-Meth *is shown through application of the inductive hypothesis and through the use of the definition for* $\varphi_0 \oplus \varphi_1$ *(where* φ_0 *is used for the receiver and* φ_1 *for the argument), which ensures a valid* φ'*.*

The cases for C-Wk-Read *and* C-Wk-Write *consume an invocation from* $\overline{ef}$*, while those for* C-Wk-Read-Body *and* C-Wk-Write-Body *replace such invocations.*

The case for C-Wk-New *is straightforward (care must be taken with heaps). The case for* C-Wk-New-Begin *has two sub-cases which depend on the number of fields defined in the class of the object being considered: the initialisation*

chord will result in either a set sequence of initial invocations to field-encoding methods (when one or more fields are defined) or directly result in returning the newly-created object (when no fields are defined); accordingly, the resulting configurations are shown to correspond with C-Wk-New-Body *in the former case and* C-Wk-Val *in the latter case.*

Finally, the case for C-Wk-New-Body *also has two sub-cases depending on the number of initial invocations remaining in the execution of the initialisation chord: if a single invocation remains then the current expression becomes the address of the object being considered, and the resulting configurations are shown to correspond through the use of* C-Wk-Void *and* C-Wk-Val*; if, however, two or more invocations remain then the same* C-Wk-New-Body *judgement is applied again, with* r *having increased by one.* □

We now use weak correspondence to show completeness. Theorem 7 says that for corresponding ^{f}SCHOOL and SCHOOL configurations, if the SCHOOL configuration makes one evaluation step, then correspondence is preserved in one of two ways: either the SCHOOL configuration evaluated into one of the intermediate steps and hence still corresponds with the initial ^{f}SCHOOL configuration, or the ^{f}SCHOOL configuration can make an evaluation step and the two resulting configurations correspond.

Theorem 7 (Completeness of The Encoding)

$$\left.\begin{array}{l} \overline{{}^{f}e}, {}^{f}h \;\approx\; \overline{e}, h \\ \overline{e}, h \;\longrightarrow\; \overline{e'}, h' \end{array}\right\} \Longrightarrow \begin{array}{l} \overline{{}^{f}e}, {}^{f}h \;\approx\; \overline{e'}, h' \quad \vee \\ \exists\, \overline{{}^{f}e'}, {}^{f}h' \;:\; \overline{{}^{f}e}, {}^{f}h \;\longrightarrow\; \overline{{}^{f}e'}, {}^{f}h' \;\wedge\; \overline{{}^{f}e'}, {}^{f}h' \;\approx\; \overline{e'}, h' \end{array}$$

Proof *By case analysis on the weak correspondence judgement applied to obtain* $\overline{{}^{f}e}, {}^{f}h \;\approx\; \overline{e}, h$*, and use of lemma 5 and definition 10. The application of the lemma yields the relationship* $\varphi, {}^{f}e, {}^{f}h \;\sim\; \varphi', {}^{f}e', {}^{f}h'$ *which determines whether the* ^{f}SCHOOL *expression has evaluated or has remained the same; the other three results satisfy the four premises of* C-Wk *(up to permutation and ignoring of leftover* voidValue *expressions, both of which are handled by* C-Wk-Perm*).* □

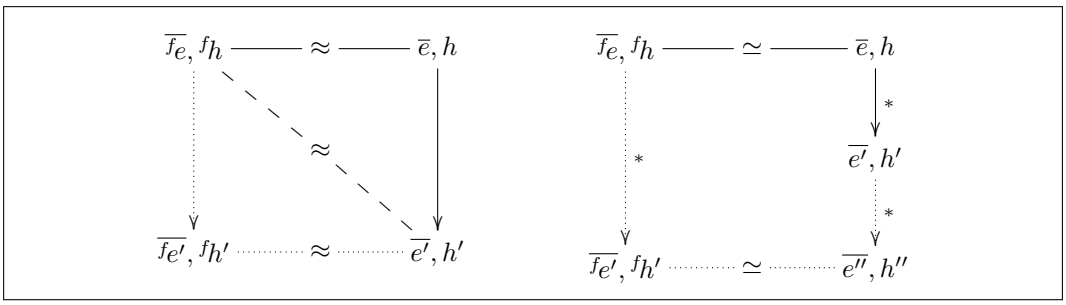

Figure 3.21: Weak completeness and strong completeness of the encoding.

The left diagram from figure 3.21 illustrates the above theorem; the solid lines represent the premises, and the dashed and dotted lines represent the first and second conclusions, respectively.

In theorem 5 we had shown soundness for the stronger version of correspondence, $\simeq$, rather than $\approx$. The converse of theorem 5 does not hold directly, but we can prove the following "diamond" property: if the SCHOOL expressions $\overline{e}$ are the encoding of the ^{f}SCHOOL expressions $\overline{{}^{f}e}$, and the expressions $\overline{e}$ evaluate in several steps to some expressions $\overline{e'}$, then the ^{f}SCHOOL expressions can be evaluated to some expressions $\overline{{}^{f}e'}$ whose encoding is the result of *further* evaluation of $\overline{e'}$ (theorem 8 below).

In order to prove the final theorem it is useful to notice that weak correspondence encompasses strong correspondence, as the latter is a special case of the former where no field-access chord is currently joined (and hence all φs are empty). In order to see this, consider the collection of weak correpsondence judgements for individual expressions which feature an empty φ: this gives correspondence between an expression ${}^{f}e$ and its direct encoding $\mathbb{C}({}^{f}e)$ (lemma 6 below), hence matching the strong correspondence judgement C-Strong for each ^{f}SCHOOL expression. Since no field-access chords are currently executing, given by definition 9, the premise of C-Strong is satisfied. However, when fields *are* being accessed, we must show (lemmas 7 and 8 below) that further evaluation always leads to corresponding configurations where no fields are being accessed any longer, and thus strong correspondence is always reached.

First, we consider weakly-corresponding expressions which do not access fields or objects, and show that the SCHOOL expression is the encoding of its corresponding ^{f}SCHOOL expression (lemma 6 below). However, during evaluation it is possible for multiple *voidValue* terms to be present; as these terms will be discarded by the implied sequencing semantics, we define the equality $=_{voidValue}$ which holds when all *voidValue* terms are removed.

Lemma 6 (Weak Correspondence When No Field Access)
$\varepsilon \vdash {}^f e, {}^f h \;\approx\; e, h \implies e =_{voidValue} \mathbb{C}({}^f e)$

Proof *By case analysis on the derivation of* $\varepsilon \vdash {}^f e, {}^f h \;\approx\; e, h$. $\square$

Second, we consider weakly-corresponding expressions which access a field or object, and show that for any such collection of expressions from the SCHOOL configuration (hence $\varphi \neq \varepsilon$), it is always the case that they will eventually evaluate so as to cease accessing the field or object, and additionally maintain weak correspondence with the result of evaluating the corresponding ^{f}SCHOOL expression (hence $\varphi = \varepsilon$). Furthermore, weak correspondence between the rest of the expressions in the configurations is preserved, and the resulting heaps correspond (lemma 7 below).

Lemma 7 (Field-Accessing Expressions Complete)

$$\left.\begin{array}{l} \varphi \vdash {}^{f}e, {}^{f}h \;\approx\; e, \overline{e}, h \\ \forall\; i \in 1..k \;:\; \varphi_i \vdash {}^{f}e_i, {}^{f}h \;\approx\; e_i, \overline{e_i}, h \\ \overline{ef} = \mathcal{F}({}^{f}h, \varphi, \varphi_1, \ldots, \varphi_k) \\ \varphi \neq \varepsilon \end{array}\right\} \Longrightarrow$$

$$\begin{array}{l} \exists\; {}^{f}e', {}^{f}h', e', \overline{ef'}, h' \;: \\ \quad {}^{f}e, {}^{f}h \;\longrightarrow\; {}^{f}e', {}^{f}h' \\ \quad e, \overline{e}, \overline{ef}, h \;\longrightarrow^{*}\; e', \overline{ef'}, h' \\ \quad \varepsilon \vdash {}^{f}e', {}^{f}h' \;\approx\; e', h' \\ \quad \forall\; i \in 1..k \;:\; \varphi_i \vdash {}^{f}e_i, {}^{f}h' \;\approx\; e_i, \overline{e_i}, h' \\ \quad \overline{ef'} = \mathcal{F}({}^{f}h', \varphi_1, \ldots, \varphi_k) \\ \quad {}^{f}h' \approx h' \end{array}$$

Proof *By structural induction on the weak correspondence judgement* $\varphi \vdash {}^{f}e, {}^{f}h \;\approx\; e, \overline{e}, h$. $\square$

We generalise the above result to weakly corresponding configurations; hence, for any SCHOOL and ^{f}SCHOOL configurations which weakly correspond, both eventually evaluate so that all fields cease to be accessed and the resulting expressions, as well as the heaps, weakly correspond (lemma 8 below).

Lemma 8 (Field-Accessing Configurations Complete)
$\overline{{}^{f}e}, {}^{f}h \;\approx\; \overline{e}, h \implies$

$$
\begin{array}{l}
\exists\ \overline{{}^{f}e'}, {}^{f}h', \overline{e'}, \overline{ef}, h' \;:\\
\quad \overline{{}^{f}e}, {}^{f}h \;\longrightarrow^{*}\; \overline{{}^{f}e'}, {}^{f}h'\\
\quad \overline{e}, h \;\longrightarrow^{*}\; \overline{e'}, \overline{ef}, h'\\
\quad \forall\ i \in 1..n \;:\; \varepsilon \vdash {}^{f}e'_i, {}^{f}h' \;\approx\; e'_i, h'\\
\quad \overline{ef} = \mathcal{F}({}^{f}h')\\
\quad {}^{f}h' \approx h'
\end{array}
$$

Proof *By induction on the number of expressions in* $\overline{{}^{f}e}$ *and repeated application of lemma 7.* $\square$

Finally, we use theorem 7 and the above lemmas to show strong completeness of the encoding (theorem 8 below).

Theorem 8 (Strong Completeness of The Encoding)

$$
\left.
\begin{array}{l}
\overline{{}^{f}e}, {}^{f}h \;\simeq\; \overline{e}, h\\
\overline{e}, h \;\longrightarrow^{*}\; \overline{e'}, h'
\end{array}
\right\} \implies
\begin{array}{l}
\exists\ \overline{{}^{f}e'}, {}^{f}h', \overline{e''}, h'' \;:\\
\quad \overline{e'}, h' \;\longrightarrow^{*}\; \overline{e''}, h'' \wedge\\
\quad \overline{{}^{f}e}, {}^{f}h \;\longrightarrow^{*}\; \overline{{}^{f}e'}, {}^{f}h' \wedge\\
\quad \overline{{}^{f}e'}, {}^{f}h' \;\simeq\; \overline{e''}, h''
\end{array}
$$

Proof *We observe that* $\overline{{}^{f}e}, {}^{f}h \;\simeq\; \overline{e}, h$ *implies* $\overline{{}^{f}e}, {}^{f}h \;\approx\; \overline{e}, h$*, and hence we can repeatedly apply theorem 7 so as to obtain* $\overline{e}, h \;\longrightarrow^{*}\; \overline{e'}, h'$ *and* $\overline{{}^{f}e}, {}^{f}h \;\longrightarrow^{*}\; \overline{{}^{f}e_0}, {}^{f}h_0$ *and* $\overline{{}^{f}e_0}, {}^{f}h_0 \;\approx\; \overline{e'}, h'$*. From lemma 8 we can now obtain* $\overline{e'}, h' \;\longrightarrow^{*}\; \overline{e''}, h''$ *and* $\overline{{}^{f}e_0}, {}^{f}h_0 \;\longrightarrow^{*}\; \overline{{}^{f}e'}, {}^{f}h'$ *and* $\overline{{}^{f}e'}, {}^{f}h' \;\approx\; \overline{e''}, h''$ *where all* φ*s are empty. Finally, from lemma 6 we obtain* $\overline{{}^{f}e'}, {}^{f}h' \;\simeq\; \overline{e''}, h''$*.* $\square$

The right diagram from figure 3.21 illustrates the above theorem; solid lines represent the premises and dotted lines represent the conclusions.

3.6 Summary and Conclusions

In this chapter we began with a generalised definition of a chord which is similar to Polyphonic C♯. We then presented SCHOOL through its syntax, operational semantics, and type system. We showed type safety and defined termination and deadlock, and subsequently showed progress. We briefly compared SCHOOL to Polyphonic C♯ and noticed that SCHOOL is a more general model of chords. We then extended SCHOOL with mutable fields, resulting in ^{f}SCHOOL, and devised an encoding of ^{f}SCHOOL programs into SCHOOL programs which use only chords to encode fields; finally we showed that the encoding is sound and complete, and thus concluded that fields are not a necessary construct of chorded languages, as they do not increase the expressive power of the language.

SCHOOL turns out to be small (only four evaluation rules and one permutation rule), and simple (a term rewriting system and a heap). Although the heap only contains objects, consisting of just their type, this does not limit the encapsulation of state: as we have seen with fields in section 3.4, a "thread-safe", or mutually exclusive, mechanism for encoding mutable fields is straightforward. Furthermore, the non-deterministic ordering of method invocations enables us to encode such "method invocation queues" implicitly using top-level asynchronous invocations, rather than including an additional queue construct in the heap.

In the concluding chapter we will briefly look at the possibility of including and encoding other typical object-oriented concurrency constructs such as explicit thread life-cycles and re-entrant monitors; we believe that the size and simplicity of SCHOOL, as well as our methodology of showing equivalence between ^{f}SCHOOL and an encoding of fields in SCHOOL in section 3.5 will assist in such future work.

Chapter 4

Weak and Strong Fairness for Chorded Languages [1]

4.1 Introduction

We present abstract schedulers for SCHOOL which ensure weak and strong process fairness. These schedulers aim to provide a foundation for realising concrete schedulers which extend the guarantees of weak and strong fairness in useful ways; in this respect, the schedulers constrain the underlying language execution in a minimal way, leaving low-level selection unspecified (non-deterministic choice in terms of SCHOOL evaluation rules).

We first identify the concurrent components of SCHOOL and the granularity of scheduling decisions, which results in an appropriate labelling scheme for execution traces. This will result in the notions of freshness and liveness of labels. Furthermore, the liveness behaviours of executions are studied, as are the worst-case behaviours under strongly-fair schedulers.

[1] The work presented in this chapter is original to this thesis.

Structure of This Chapter

Section 4.2 presents a labelling scheme for SCHOOL, resulting in ^{l}SCHOOL, along with the augmented semantics and notions of label freshness and liveness, and worst-case liveness behaviours of executions; section 4.3 gives definitions of weak and strong fairness, along with examples of inadmissible executions, in terms of ^{l}SCHOOL; section 4.4 presents an abstract weakly-fair scheduler; finally section 4.5 presents an abstract strongly-fair scheduler, along with worst-case behaviours of executions under the scheduler.

4.2 Labelled SCHOOL

The granularity level of SCHOOL 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. Our presentation of fairness then requires referencing of individual expressions throughout executions, and a straightforward way to achieve this is to employ a labelled transition system, resulting in ^{l}SCHOOL. Individual expressions are annotated with unique labels, and transitions are annotated with collections of labels which correspond to those expressions participating in the evaluation resulting in the transition.

The use of labels thus enables us to observe execution traces, and classify these as admissible or inadmissible under particular strengths of fairness. In order to classify original SCHOOL executions as admissible or inadmissible, we must ensure that ^{l}SCHOOL executions correspond to SCHOOL executions.

4.2.1 Labels

Labels are used to identify, and refer to, expressions within a configuration. Accordingly, each expression is annotated with a distinct label which forms part of the runtime entities of the language. Expressions which are ground

values, and hence not considered during future evaluation steps, are each labelled with a special *empty* label.

Definition 11 (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,* $\mathcal{L}^\varepsilon$*; 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 12 (Labelled Configuration)
A configuration, $e_1, \ldots, e_n, h$*, is* labelled *by annotating each expression with a label:*

$$\mathit{labelled}\,(e_1, \ldots, e_n) = e_1^{l_1}, \ldots, e_n^{l_n}$$

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: $\overline{e^l}, h$*.*

From the previous definition we accept that, for a uniquely labelled configuration, refering to the l_i expression is equivalent to referring to the i'th expression. Furthermore, this definition has a straightforward converse (definition 13 below) with which we can retrieve the original unlabelled form from any labelled configuration.

Definition 13 (Unlabelled Configuration)
$\mathit{unlabelled}\left(e_1^{l_1}, \ldots, e_n^{l_n}\right) = e_1, \ldots, e_n$

Consequently, labelling and unlabelling of configurations does not change their underlying form (lemma 9 below); this enables us to derive SCHOOL executions from ^{l}SCHOOL executions by dropping all references to labels.

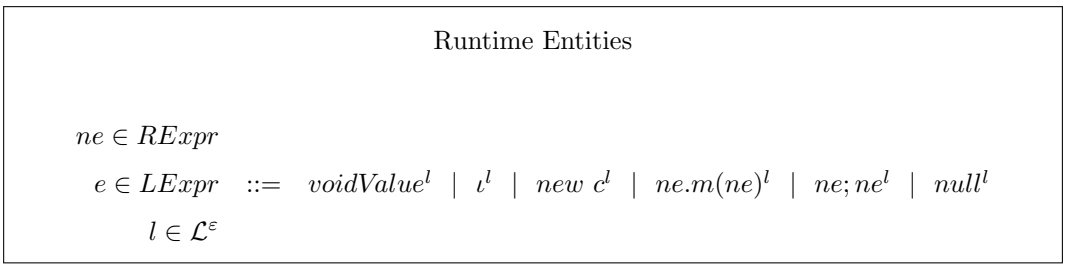

Figure 4.1: ^{l}SCHOOL overview.

Lemma 9 (Labelling Preserves Unlabelled Form)
unlabelled $(\mathit{labelled}\,(e_1, \ldots, e_n)) = e_1, \ldots, e_n$

Proof *Straightforward from definitions 12 and 13.* □

4.2.2 Labelled Semantics

The base language SCHOOL is extended with runtime expressions featuring label annotations, as in figure 4.1, forming the language ^{l}SCHOOL; only top-level expressions, *LExpr*, are annotated, and so all sub-expressions come directly from the SCHOOL runtime expressions *RExpr* and have no labels. Figure 4.2 presents the evaluation rules for ^{l}SCHOOL. The form of evaluation in ^{l}SCHOOL (definition 14 below) incorporates the notion of *participating* labels: at each step there is a set of labels which refer to the expressions affected by the rule used to perform that step. Furthermore, ground values become annotated with the empty label, and each changed or new expression becomes annotated with a distinct, *fresh* label.

Definition 14 (Labelled Evaluation Step)
A labelled evaluation step has the following form:

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

where an initial uniquely labelled configuration, $e_1^{l_1}, \ldots, e_n^{l_n}, h$*, evaluates to a final uniquely labelled configuration,* $e_1^{'\,l'_1}, \ldots, e_m^{'\;l'_m}, h'$*, through a non-empty set*

$$\frac{h(\iota) \text{ is undefined} \quad l' \text{ is fresh}}{E[new\ c]^l, h \xrightarrow{\{l\}} E[\iota]^{l'}, h[\iota \mapsto c]}\ \textsc{New} \qquad \frac{l' \text{ is fresh}}{E[z;e]^l, h \longrightarrow E[e]^{l'}, h}\ \textsc{Seq}$$

$$\frac{h(\iota){=}c \quad m \in \bigcup_{\chi \in P\downarrow_3(c)} \chi\downarrow_2 \quad E[\centerdot] \neq [\centerdot] \quad l', l'' \text{ are fresh}}{E[\iota.m(v)]^l, h \xrightarrow{\{l\}} E[voidValue]^{l'}, \iota.m(v)^{l''}, h}\ \textsc{Async}$$

$$\frac{\begin{array}{l} h(\iota){=}c \quad (m, \{m_1, \ldots, m_k\}, e) \in P\downarrow_3(c) \quad l'_0 \text{ fresh} \\ \forall\ i \in 1..k\ :\ \exists\ l'_i \in \mathcal{L}^{\varepsilon}\ :\ l'_i = \begin{cases} \varepsilon & \text{if } E_i[\centerdot] = [\centerdot] \\ l \in \mathcal{L}, l \text{ fresh} & \text{otherwise} \end{cases} \end{array}}{\begin{array}{l} E[\iota.m(v)]^{l_0}, E_1[\iota.m_1(v_1)]^{l_1}, \ldots, E_1[\iota.m_k(v_k)]^{l_k}, h \xrightarrow{\{l_0, l_1, \ldots, l_k\}} \\ \quad E[e[{}^{\iota}/_{this}, {}^{v}/_{m\_x}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]]^{l'_0}, E_1[voidValue_1]^{l'_1}, \ldots, E_k[voidValue_k]^{l'_k}, h \end{array}}\ \textsc{Join}$$

$$\frac{\begin{array}{l} h(\iota){=}c \quad (\varepsilon, \{m_1, \ldots, m_k\}, e) \in P\downarrow_3(c) \quad l \text{ fresh} \\ \forall\ i \in 1..k\ :\ \exists\ l'_i \in \mathcal{L}^{\varepsilon}\ :\ l'_i = \begin{cases} \varepsilon & \text{if } E_i[\centerdot] = [\centerdot] \\ l' \in \mathcal{L}, l' \text{ fresh} & \text{otherwise} \end{cases} \end{array}}{\begin{array}{l} E_1[\iota.m_1(v_1)]^{l_1}, \ldots, E_k[\iota.m_k(v_k)]^{l_k}, h \xrightarrow{\{l_1, \ldots, l_k\}} \\ \quad E_1[voidValue_1]^{l'_1}, \ldots, E_k[voidValue_k]^{l'_k}, E[e[{}^{\iota}/_{this}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]]^{l}, h \end{array}}\ \textsc{Strung}$$

$$\frac{\overline{e^l} \cong \overline{e''^{l''} e'''^{l'''}} \quad \overline{e''^{l''}}, h \xrightarrow{\mu} \overline{e'''^{l'''}}, h' \quad \overline{e'''^{l'''} e''''^{l''''}} \cong \overline{e'^{l'}}}{\overline{e^l}, h \xrightarrow{\mu} \overline{e'^{l'}}, h'}\ \textsc{Perm}$$

Figure 4.2: ^{l}SCHOOL operational semantics.

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 $m \geqslant n$. We abbreviate a labelled evaluation step as: $\overline{e^l}, h \xrightarrow{\mu} \overline{e'^{l'}}, h'$.*

Definition 15 (Labelled Execution)

A (possibly infinite) labelled execution has the form:

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

which consists of a sequence of evaluation steps, indexed in $\mathbb{N}$.

We obtain the labels of labelled configuration by ignoring the underlying expressions (definition 16 below).

Definition 16 (Labels of a Configuration)
$labels\left(e_1^{l_1}, \ldots, e_n^{l_n}\right) = \{l_1, \ldots, l_n\}$

When considering which labels participate in an evaluation step it is useful to differentiate between ground values, which can never participate in an execution, and the rest of the expression, which *may* participate in an execution. Hence, we define the *active* labels of a labelled configuration, $e_1^{l_1}, \ldots, e_n^{l_n}, h$, as the non-empty labels (definition 17 below).

Definition 17 (Active Labels)
$active\left(e_1^{l_1}, \ldots, e_n^{l_n}\right) = \{l_i \in labels\left(e_1^{l_1}, \ldots, e_n^{l_n}\right) : l_i \neq \varepsilon\}$

The freshness of a label is a property of the entire execution sequence up to the first appearance of the label (definition 18 below).

Definition 18 (Freshness of Labels)

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

l is fresh at i

The participating labels, μ, form a set (lemma 10 below) because the evaluation rules are defined only for uniquely labelled configurations, and hence each label appears exactly once. Furthermore, the participating labels cannot be empty (lemma 11 below) because each rule changes at least one expression

from the initial configuration. Also, the empty label cannot be a participating label (lemma 12 below) as ground values never contribute to an evaluation. Finally, participating labels are consumed (lemma 13 below). These results show that application of the ^{l}SCHOOL evaluation rules results in labelled evaluation steps as per definition 14 above.

Lemma 10 (Participating Labels Form A Set)
$\overline{e^{l}}, h \xrightarrow{\{l_1,\ldots,l_n\}} \overline{e'^{l'}}, h' \implies \forall\ i, j \in 1..n\ :\ l_i = l_j \implies i = j$

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

Lemma 11 (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 derviation of* $\overline{e^{l}}, h \xrightarrow{\mu} \overline{e'^{l'}}, h'$*; case analysis on the last evaluation rule applied.* □

Lemma 12 (Empty Labels Never Participate)
$\overline{e^{l}}, h \xrightarrow{\mu} \overline{e'^{l'}}, h' \implies \varepsilon \notin \mu$

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

Lemma 13 (Participating Labels are Consumed)
$\overline{e^{l}}, h \xrightarrow{\mu} \overline{e'^{l'}}, h' \implies \forall\ l \in \mu\ :\ l \notin labels\left(\overline{e'^{l'}}\right)$

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

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

Listing 4.1: Example of class for labelled semantics.

$$\overline{e_0^{l_0}}, h_0 \equiv \quad \iota.f()^{l_1}, \iota.g()^{l_2}, \iota.a()^{l_3}, \iota.a()^{l_4}, h$$

$$\xrightarrow[\mu_0=\{l_1,l_3\}]{\textsc{Join}}$$

$$\overline{e_1^{l_1}}, h_1 \equiv \quad E[\iota.b()]^{l_5}, \iota.g()^{l_2}, voidValue^{\varepsilon}, \iota.a()^{l_4}, h$$

$$\xrightarrow[\mu_1=\{l_5\}]{\textsc{Async}}$$

$$\overline{e_2^{l_2}}, h_2 \equiv \quad E[\,]^{l_6}, \iota.g()^{l_2}, voidValue^{\varepsilon}, \iota.a()^{l_4}, \iota.b()^{l_7}, h$$

$$\xrightarrow[\mu_2=\{l_2,l_7\}]{\textsc{Join}}$$

$$\overline{e_3^{l_3}}, h_3 \equiv \quad E[\,]^{l_6}, E[print\ ''Hi'']^{l_8}, voidValue^{\varepsilon}, \iota.h()^{l_4}, voidValue^{\varepsilon}, h$$

Figure 4.3: An execution of the `LabelledExample` class in ^{l}SCHOOL.

Example

We consider an example of a labelled execution in figure 4.3. We use the class `LabelledExample` from listing 4.1 which features 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 (the configuration indexed by 0) there exists a sole object of class `LabelledExample` at address ι, and four invocations of methods `f`, `g` and two of `a` respectively.

The first step of evaluation consists of a join of the first chord and hence an application of 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 by the application of the rule, and thus do not appear in the next configuration (indexed by 1). The consuming of the asynchronous invocation of `a` results in its replacement with *voidValue*, and its annotation with the empty label, ε. The body of the chord now forms a new expression, and is annotated with the newly created label l_5.

The label l_2, which did not participate in the evaluation step, is preserved and 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 hence the label remains throughout. As with indexed labels which do not participate, the *voidValue* expressions with their empty label annotations are preserved by the evaluation rules.

Correspondence

The introduction of labels to the runtime entities of SCHOOL does not change the underlying semantics of the language; furthermore, each transition is uniquely described by the participating labels, and hence the evaluation rule and the labels involved can be determined. Therefore, it is straightforward to show a direct correspondence between ^{l}SCHOOL and SCHOOL evaluation steps (lemma 14 below) and executions (theorem 9 below).

Lemma 14 (Labelled and Unlabelled Steps Correspond)

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

Proof *Straightforward by application of lemma 9 and case analysis on the evaluation rule used.* □

Theorem 9 (Labelled and Unlabelled Executions Correspond)

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

Proof *Straightforward induction on the length of the execution and application of lemma 14 for the correspondence of each step.* □

Properties of Labelled Execution

By inspection of the evaluation rules we know that the labels of a configuration remain finite (lemma 15 below), and there is an upper bound on the creation

of new expressions, and hence new labels (lemma 16 below). The rules New and Perm maintain the number of labels. The rule Async always introduces a new expression and its associated new label into the configuration. The rule Join must consume a synchronous invocation, and depending on whether the selected chord contains asynchronous methods or not, and whether the asynchronous invocations consumed were within an evaluation context or not, application of the rule may maintain or reduce the number of labels in the configuration. The rule Strung introduces a new expression and its associated new label, however, it may consume more than one asynchronous invocation, so its application may reduce the total number of labels in a configuration.

Lemma 15 (Labels Remain Finite)

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

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

Lemma 16 (Upper Bound On Creation Of Labels)

$$\overline{e^{l}}, h\ \xrightarrow{\mu}\ \overline{e'^{l'}}, h'\ \Longrightarrow\ \exists\ \lambda \in \mathbb{N}\ :\ |active\left(\overline{e'^{l'}}\right)| \leqslant |active\left(\overline{e^{l}}\right)| + \lambda$$

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

4.2.3 Liveness

The *live* labels of a configuration, $e_1^{l_1}, \ldots, e_n^{l_n}, h$, are those labels which can participate in the next 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 (definition 19 below).

Definition 19 (Live Labels)

$$live\left(e_1^{l_1},\ldots,e_n^{l_n}\right) = \{l_i \in active\left(e_1^{l_1},\ldots,e_n^{l_n}\right) :$$
$$\exists\ {e_1'}^{l_1'},\ldots,{e_m'}^{l_m'}, h' \ :\ e_1^{l_1},\ldots,e_n^{l_n}, h \xrightarrow{\mu} {e_1'}^{l_1'},\ldots,{e_m'}^{l_m'}, h' \wedge l_i \in \mu\}$$

It is possible that an evaluation rule is 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. For instance, 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* (definition 20 below).

Definition 20 (Ignoring a Label)

$$\left.\begin{array}{l} l \in live\left(\overline{e^l}\right) \\ \overline{e^l}, h \xrightarrow{\mu} \overline{e'^{l'}}, h' \wedge l \notin \mu \end{array}\right\} \implies l \text{ was ignored}$$

A label implicitly loses its liveness when it participates in a step of evaluation, as it is consumed and does not appear in any further configurations. 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 (or may regain its liveness if previously lost) due to a newly created label (such as a method invocation required for a join).

Example

We consider an example of liveness in figure 4.4. To do this we use the class `LivenessExample` from listing 4.2, similar to `LabelledExample` from listing 4.1, which now features a third chord; the second and third chord both require an invocation of `b` in order to join, and hence will compete for such an invocation. Initially (the configuration indexed by 0) there exists a sole object of

```
class LivenessExample {
  void f() & async a() { b(); }
  void g() & async b() { print "Hi"; }
  void h() & async b() { print "I feel ignored"; }
}
```

Listing 4.2: Example class for lieveness of labels.

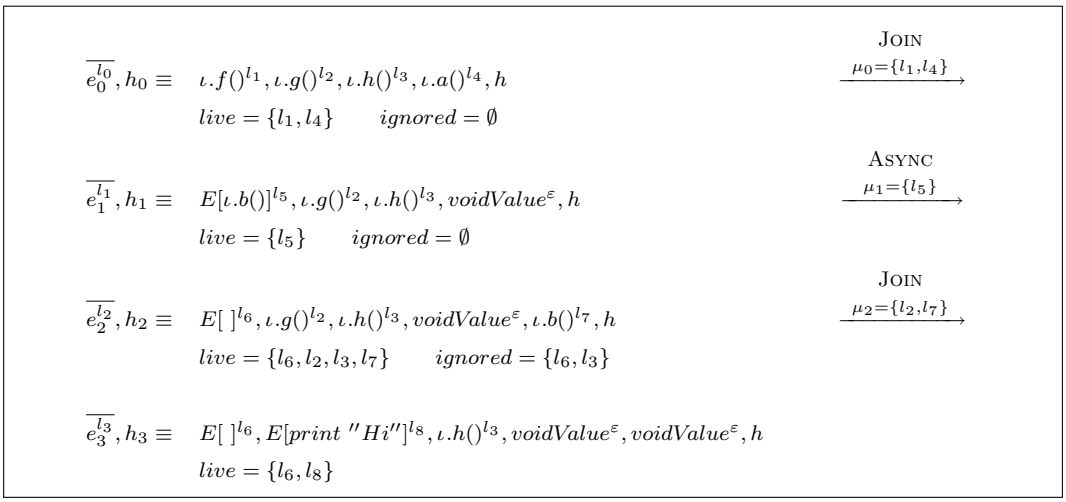

Figure 4.4: An execution of the `LivenessExample` class in ^{l}SCHOOL.

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 (at the configuration indexed by 2) both the second and third chords can join, as both their synchronous method invocations, `g` and `h`, respectively, 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`).

4.2.4 Liveness Behaviour of Labels

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

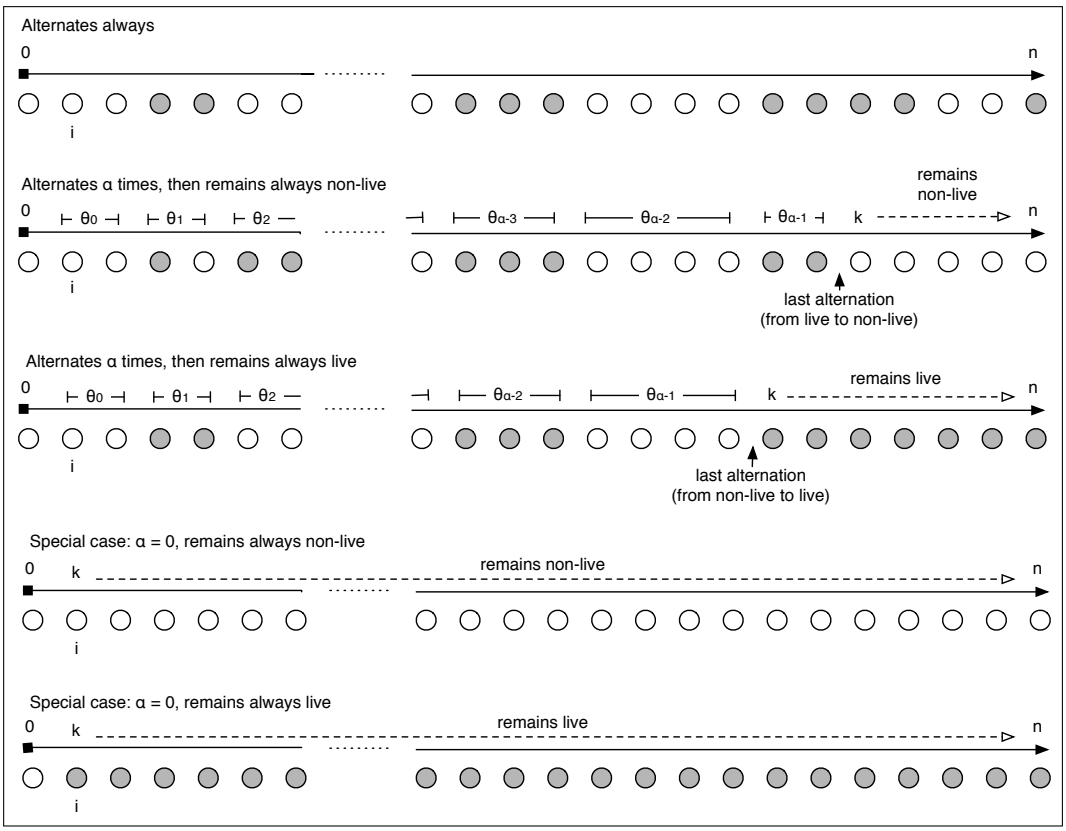

Figure 4.5: Liveness behaviour of a label: alternation.

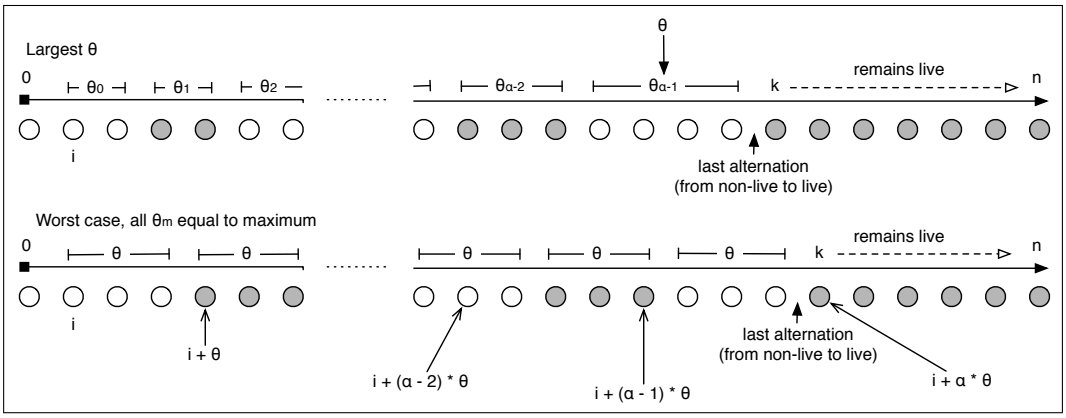

Figure 4.6: Liveness behaviour of a label: worst case for the number of configurations which occur before a label alternates for the last time.

is possible to classify liveness behaviour in terms of patterns exhibited, and accordingly make observations on the properties of these patterns.

Observation 1 (Liveness Behaviour of a Label)

For a (possibly infinite) execution sequence

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

assume a label, l, appears in the execution for the first time at configuration i, and hence $l \in labels\left(\overline{e_i^{l_i}}\right) \ \wedge \ \forall\ j \in \mathbb{N}, 0 \leqslant j < i \ : \ l \notin labels\left(\overline{e_j^{l_j}}\right)$; 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 $\exists\ \theta_0, \theta_1, \ldots, \theta_{\alpha-1} \in \mathbb{N}$, with θ_0 denoting the number of configurations before the first alternation, starting from the i'th configuration, and $\forall\ m, 1 \leqslant m \leqslant \alpha - 1$, 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: $\forall\ k \geqslant \sum_{m=0}^{\alpha-1} \theta_m + 1 \ : \ l \notin live\left(\overline{e_k^{l_k}}\right)$, or*

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

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

Figure 4.5 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 $\alpha = 0$; white circles indicate a configuration at which the label is not live, and grey cricles indicate a configuration at which the label is live.

Observation 2 (Largest θ and Worst Case Behaviour)

For a (possibly infinite) execution sequence

$l \in live\left(\overline{e_i^{l_i}}\right)$	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 4.1: Summary of worst-case calculations for number of steps which occur when ignoring a label is maximised.

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

assume a label, l, appears in the execution for the first time at configuration i, and hence $l \in labels\left(\overline{e_i^{l_i}}\right) \ \wedge \ \forall\ j \in \mathbb{N}, 0 \leqslant j < i \ : \ l \notin labels\left(\overline{e_j^{l_j}}\right)$; 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: $\theta \equiv \max\{\theta_0, \ldots, \theta_{\alpha-1}\}$. The first example execution of figure 4.6 illustrates a θ.

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 4.6 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.

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

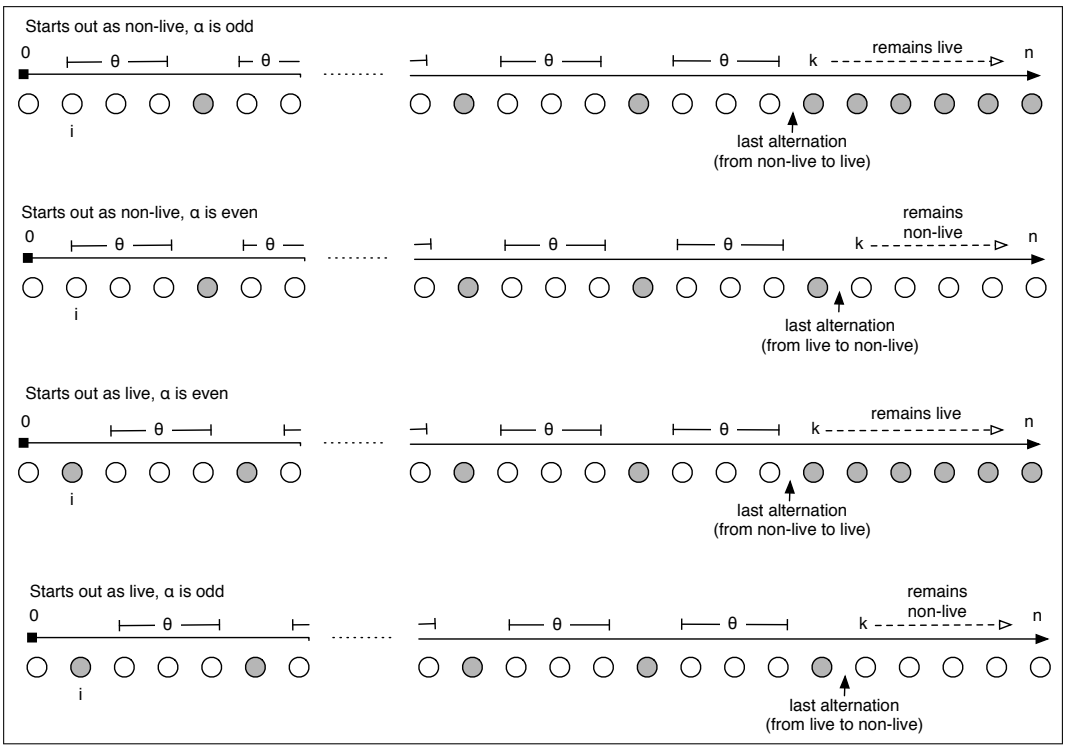

Figure 4.7: Liveness behaviour of a label: maximising the number of times a label is ignored.

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 4.7 illustrates the four cases which arise, and table 4.1 summarises the calculations.

4.3 Definitions of Weak and Strong Fairness

From definition 20 we know that a label is ignored only when it is live and does not participate in the current step of evaluation. Furthermore, from lemma 19 we know that only active labels can become live. Hence, the following definitions of fairness are stated in terms of live labels, and all empty labels are ignored.

```
class WeakFairnessExample {
  void f() & async a() { b(); a(); f(); }
  void g() & async b() { print "Oh, dear..."; }
  void h() & async a() { print "Help!"; }
  void k() & async c() { print "Ha!"; }
}
```

Listing 4.3: Example class for weak fairness.

4.3.1 Definition of Weak Fairness

Definition 21 (Weak Fairness)

An execution is weakly-fair iff no process remains ignored continuously. In other words, no process remains live continuously. [21, 45, 34]

By definition 21, an execution is weakly-fair when no labels are ignored continuously. Only live labels can be ignored, and therefore no label may be live continuously. Consequently, 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 22 (Weakly-Fair Execution)

$$\overline{e_0^{l_0}}, h_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1 \xrightarrow{\mu_1} \overline{e_2^{l_2}}, h_2 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow{\mu_n} \ldots \Longrightarrow$$
$$\forall\, i \in \mathbb{N} \;:\; \forall\, l \in \mathit{labels}\left(\overline{e_i^{l_i}}\right) \;:\; \exists\, k \geqslant i \;:\; l \notin \mathit{live}\left(\overline{e_k^{l_k}}\right)$$

Example

We consider an example execution inadmissible under weak fairness. To do this we use the class from listing 4.3 and the initial configuration from figure 4.8. 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

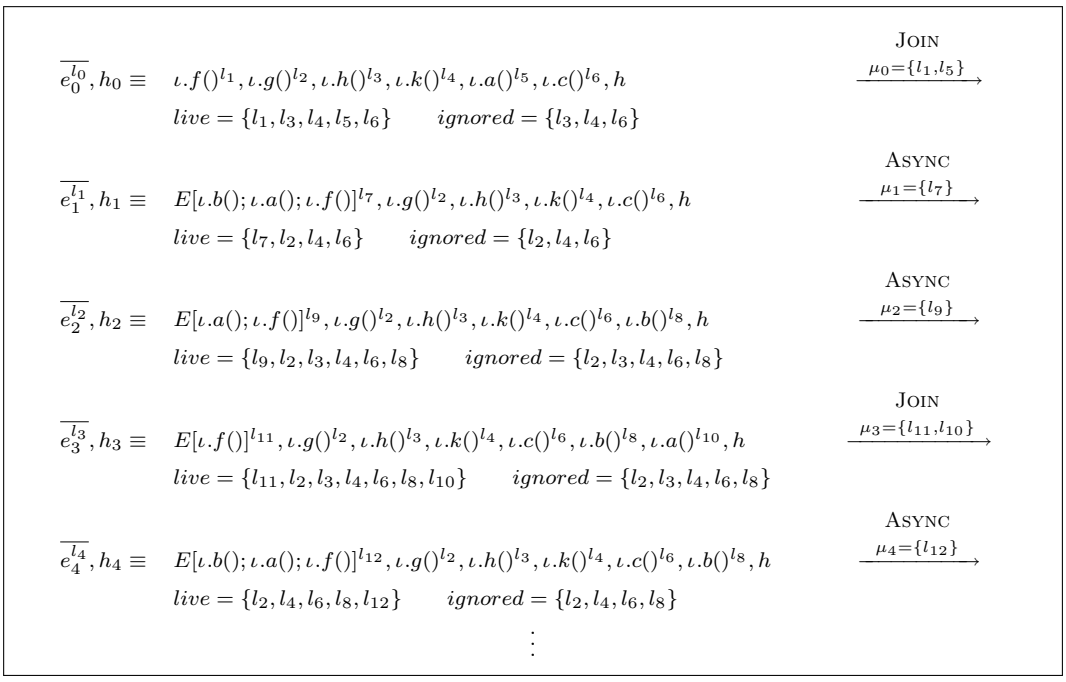

Figure 4.8: An execution inadmissible under weak fairness.

asynchronous method `b`, and finally the fourth chord requires the synchronous method `k` to join with the asynchronous method `c`. Initially (the configuration indexed by 0), 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 (indexed by 1) 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. Finally, we notice that the fourth chord has been ignored once more (labels l_4 and l_6).

At this point (configuration 2) l_3 becomes live again, as it is possible to apply the JOIN rule with the `a` from the evaluation context of l_{11}; 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.

We notice that l_2 became live in the second step and has also remained live at each further step; hence the execution, after the second step, is not weakly fair for l_2 either. However, we see that l_3 loses and regains its liveness every three steps, and thus the execution satisfies weak fairness for this label (weak fairness cannot prohibit a label being ignored infinitely often - this is the problem dealt with by strong fairness below). In order for the execution to become weakly fair, l_2 must at some point participate with one of the invocations of `b` which accumulate (then all other invocations of `b` lose their liveness instantly), and also the fourth chord must join (l_4 and l_6). Thus we expect that a weakly-fair execution sequence will eventually print both "Ha!" and "Oh, dear...", although "Help!" has no guarantee of ever printing.

4.3.2 Definition of Strong Fairness

Definition 23 (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. [21, 45, 34]

By definition 23, an execution is strongly-fair when no labels are ignored infinitely often. Only live labels can be ignored, and therefore no label may be live infinitely often. Consequently, every label must cease to be live after a finite number of steps, and never regain its liveness. The following definition requires

```
class StrongFairnessExample {
  void f() & async a() { b(); f(); }
  void f() & async b() { b(); f(); }
  async b() { a(); }
  void g() & async a() { print "Help"; }
}
```

Listing 4.4: Example class for strong fairness.

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

Definition 24 (Strongly-Fair Execution)

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

Example

We consider an example execution inadmissible under strong fairness. To do this we use the class from listing 4.4 and the initial configuration from figure 4.9. 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 (the configuration indexed by 0), 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, as there exists an invocation of `a`, labelled l_3, with which it can join and contribute to the first move. 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

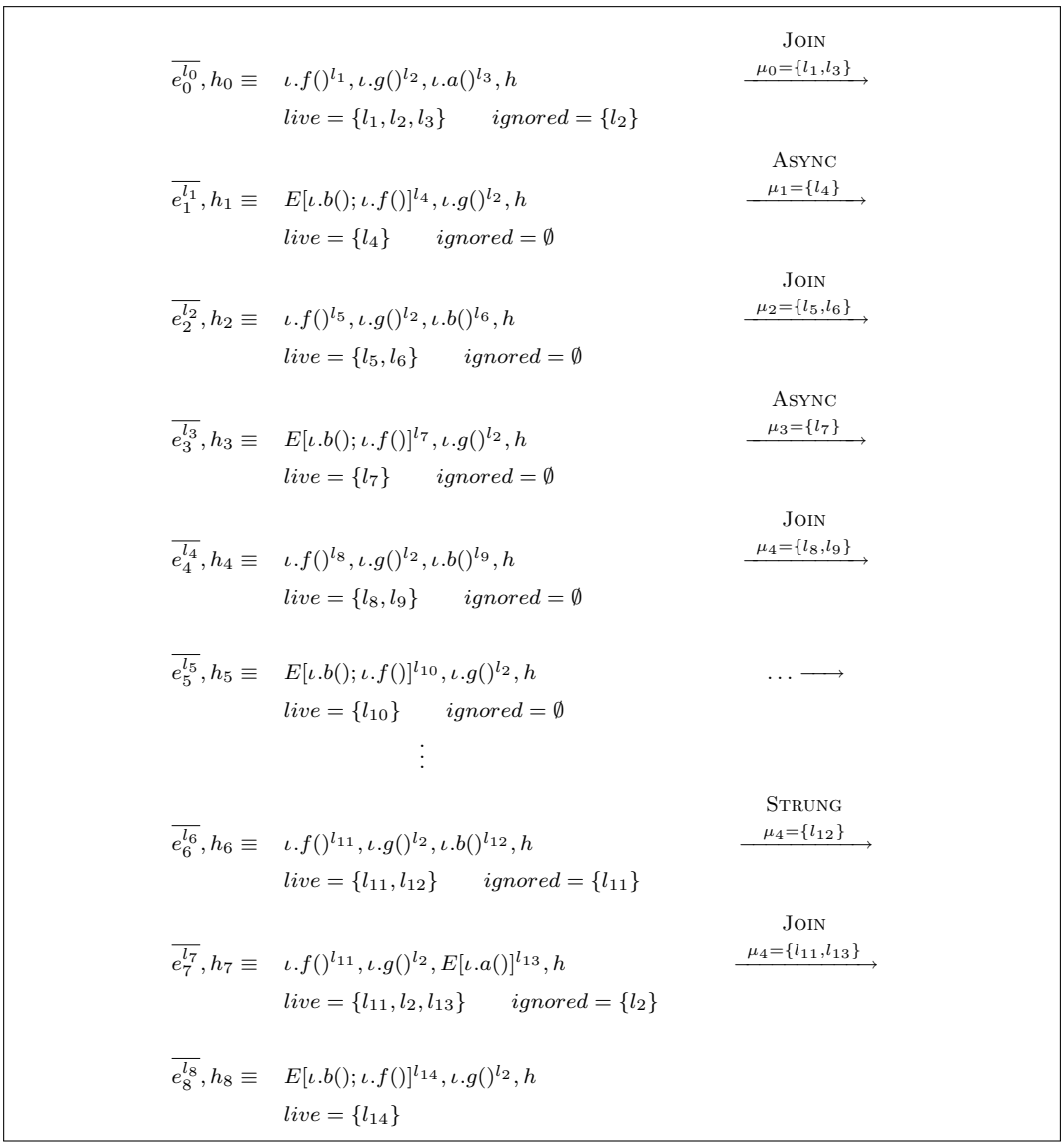

Figure 4.9: An execution inadmissible under strong fairness.

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. In the example, the third chord eventually joins (through the STRUNG rule) as in the configuration indexed by 7. But now l_2 has regained its liveness, as it is possible to join the

fourth chord with l_{13}. It so happens that in the example l_2 is ignored a second time, as the first chord joins (labels l_{11} and l_{13}).

It is clear that if this pattern were to continue forever, then l_2 would 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.

4.4 Weak Fairness

We describe a mechanism which, with an appropriate implementation, realises the weak fairness constraints for ^{l}SCHOOL. We show how the original example from section 4.3.1 of an execution inadmissible under weak fairness is altered by imposing the constraints so as to become admissible.

Some aspects of the mechanism remain unspecified, resulting in the semantics being non-deterministic with regards to choice of labels and evaluation rules. It is up to the implementation of a weakly-fair scheduler to specify these selection functions.

4.4.1 Overview

An execution sequence is weakly-fair when no label remains live throughout; accordingly, starting from any given configuration, each of the live labels appearing in that configuration must eventually lose their liveness during execution. This allows us to consider a *localised* definition of weak fairness: a sequence is *locally weakly-fair for the set of live labels in its initial configuration* (or for the *initially live labels*) only if each of the labels is not live in at least one future configuration. Once all live labels of the initial configuration have lost their liveness, we obtain a locally weakly-fair execution.

We can then proceed to obtain the next locally weakly-fair execution by

using the above final configuration as the new initial configuration. The concatenation of two locally weakly-fair execution sequences is weakly-fair, as all labels have lost their liveness in at least one of the two concatenated sequences. Also, it is straightforward to show that 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.

It is possible that new labels are created at intermediate configurations of locally weakly-fair execution sequences, and these labels may or may not remain live throughout the *remainder* of the locally weakly-fair execution sequence. In the latter case, the execution satisfies weak fairness as the labels lose their liveness. In the former case, the labels will also be live in the final configuration; but then these labels will be included in the set of live labels of the initial configuration of the *next* locally weakly-fair execution, and hence also will eventually lose their liveness.

Since executions can have an infinite length, a weakly-fair execution is then defined as *the maximal sequence of concatenated locally weakly-fair executions*. As we have seen from section 4.3.1 above, there is no finite upper limit on the number of steps before a label loses its liveness, and hence there is no finite upper limit on the length of a locally weakly-fair execution sequence.

One way to generate locally weakly-fair executions is to keep track of labels which have lost their liveness either because they have participated or because they have lost their ability to join with other labels which have been consumed; we consider these labels as *serviced*, and aim to service the remainder of the initial configuration's live labels. 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.

$$\frac{\begin{array}{l}\overline{e^l}, h \xrightarrow{\mu} \overline{e'^{l'}}, h' \\ L' \text{ largest subset of } live\left(\overline{e^l}\right) : \\ \quad L \cap L' = \emptyset \quad \wedge \quad L' \cap live\left(\overline{e'^{l'}}\right) = \emptyset\end{array}}{\overline{e^l}, h, L \overset{\mu}{\Longrightarrow} \overline{e'^{l'}}, h', L \cup L'} \textsc{Weak}$$

Figure 4.10: Weakly-fair selection rule.

4.4.2 Weakly-Fair Execution

We define a weakly-fair execution (definition 27 below) as the maximal sequence of locally weakly-fair execution sequences (definition 26 below); a locally weakly-fair execution sequence consists of weakly-fair evaluation steps (definition 25 below), which are obtained through the Weak selection rule of figure 4.10.

Definition 25 (Weakly-Fair Evaluation Step)
A weakly-fair evaluation step has the following form:

$$e_1^{l_1}, \ldots, e_n^{l_n}, h, L \overset{\mu}{\Longrightarrow} e_1^{'\,l'_1}, \ldots, e_m^{'\,l'_m}, h', L'$$

where an initial labelled configuration, $e_1^{l_1}, \ldots, e_n^{l_n}, h$*, evaluates to a final labelled configuration,* $e_1^{'\,l'_1}, \ldots, e_m^{'\,l'_m}, h'$*, 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* $m \geqslant n$*. We abbreviate a weakly-fair evaluation step as:* $\overline{e^l}, h, L \overset{\mu}{\Longrightarrow} \overline{e'^{l'}}, h', L'$*. A (possibly infinite) sequence of weakly-fair evaluation steps thus has the form:*

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

The rule Weak allows us to select any ^{l}SCHOOL 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 (either by participating or by losing their ability to join); 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.

WEAK does not *force* the selection of an ^{l}SCHOOL evaluation rule such that initially live labels are serviced, rather, it keeps track of serviced labels and can be used as a *suggestion* of which labels should lose their liveness in order for the weak fairness constraint to be satisfied; the selection of evaluation rule hence remains unspecified and is the responsibility of the scheduler implementation. However, 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.

Definition 26 (Locally Weakly-Fair Execution)

$\overline{e^l}, h \underset{L}{\overset{M}{\Longrightarrow}} \overline{e'^{l'}}, h'$ iff

$$\begin{array}{l} \exists\ k \in \mathbb{N}\ : \\ \quad \overline{e_0^{l_0}}, h_0, L_0 \overset{\mu_0}{\Longrightarrow} \overline{e_1^{l_1}}, h_1, L_1 \overset{\mu_1}{\Longrightarrow} \ \ldots\ \overset{\mu_{k-1}}{\Longrightarrow} \overline{e_k^{l_k}}, h_k, L_k \\ \quad \text{where} \quad \overline{e^l}, h \equiv \overline{e_0^{l_0}}, h_0 \quad \text{and} \quad \overline{e'^{l'}}, h' \equiv \overline{e_k^{l_k}}, h_k \\ \quad \text{and} \quad M \equiv \mu_0 \circ \mu_1 \circ \ldots \circ \mu_{k-1} \quad \text{and} \quad L \equiv L_k \\ \wedge \quad live\left(\overline{e^l}\right) \ \subseteq \ L \\ \wedge \quad \exists\ m \in \mathbb{N}\ :\ |labels\left(\overline{e_0^{l_0}}\right)| = m \\ \wedge \quad L_0 = \emptyset \end{array}$$

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 (see lemma 16 from section 4.2.2 above), and hence the number of labels which can participate and become added to each L, the size of L_k has no upper bound as there is no upper bound on the length k.

Definition 27 (Weakly-Fair Execution)

$\overline{e^l}, h \;\Rrightarrow\; \overline{e'^{l'}}, h'$ iff

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

$$\overline{e_0^{l_0}}, h_0 \;\underset{L_0}{\overset{M_0}{\Longrightarrow}}\; \overline{e_1^{l_1}}, h_1 \;\underset{L_1}{\overset{M_1}{\Longrightarrow}}\; \ldots \;\underset{L_{n-1}}{\overset{M_{n-1}}{\Longrightarrow}}\; \overline{e_n^{l_n}}, h_n \;\underset{L_n}{\overset{M_n}{\Longrightarrow}}\; \ldots$$

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

4.4.3 Example

We use the original example of weak fairness with the program of listing 4.3 and the initial configuration from figure 4.8, both from section 4.3.1 above. In figure 4.11 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, where newly added labels are 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

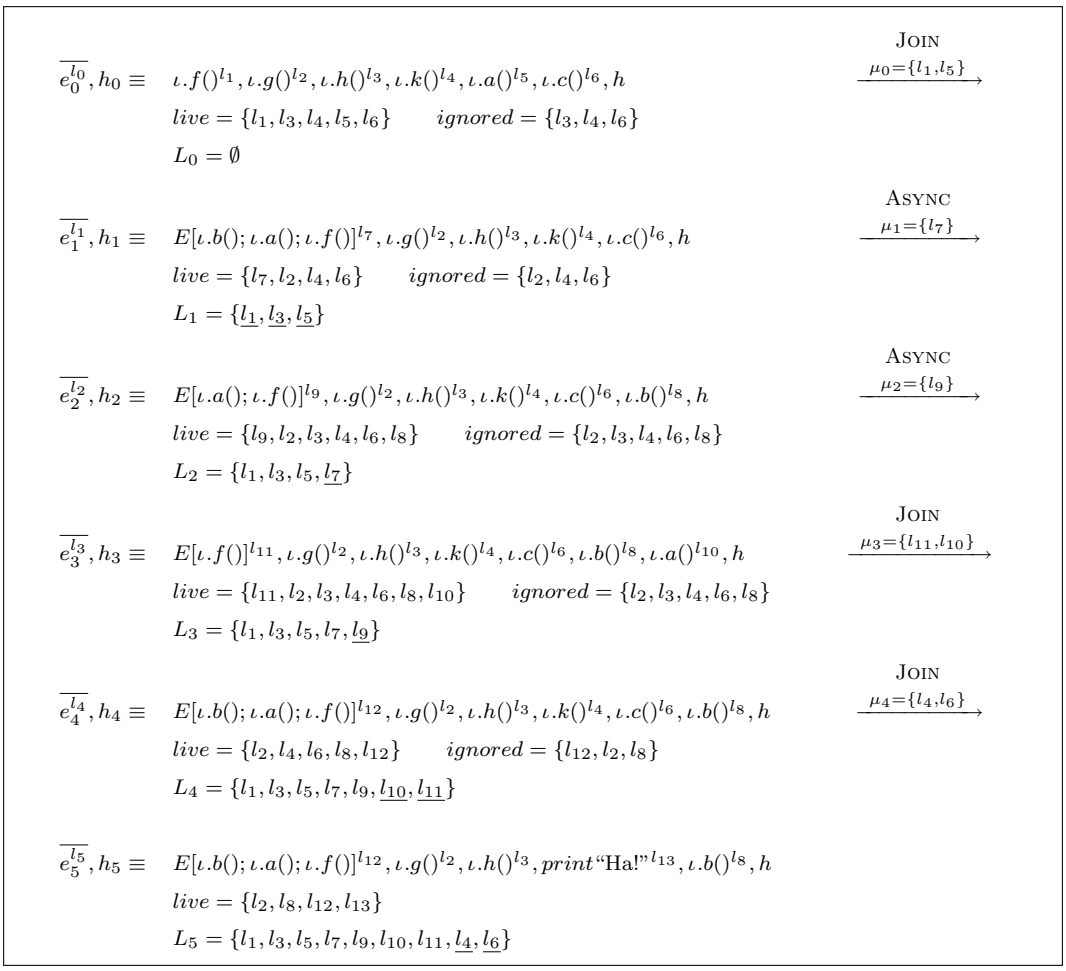

Figure 4.11: A locally weakly-fair execution.

execution sequence by aiming to service the live labels $\{l_2, l_8, l_{12}, l_{13}\}$; an example such sequence is in figure 4.12 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, equals to the set of original live labels, hence making this sequence locally weakly-fair.

Therefore, the execution sequence indexed by 0 through 5 is locally weakly-fair, written $\overline{e_0^{l_0}}, h_0 \overset{M}{\underset{L}{\Longrightarrow}} \overline{e_5^{l_5}}, h_5$, where $M = \{l_1, l_5\} \circ \{l_7\} \circ \{l_9\} \circ \{l_{11}, l_{10}\} \circ \{l_4, l_6\}$ and $L = \{l_1, l_3, l_5, l_7, l_9, l_{10}, l_{11}, l_4, l_6\}$; the execution sequence indexed by 5

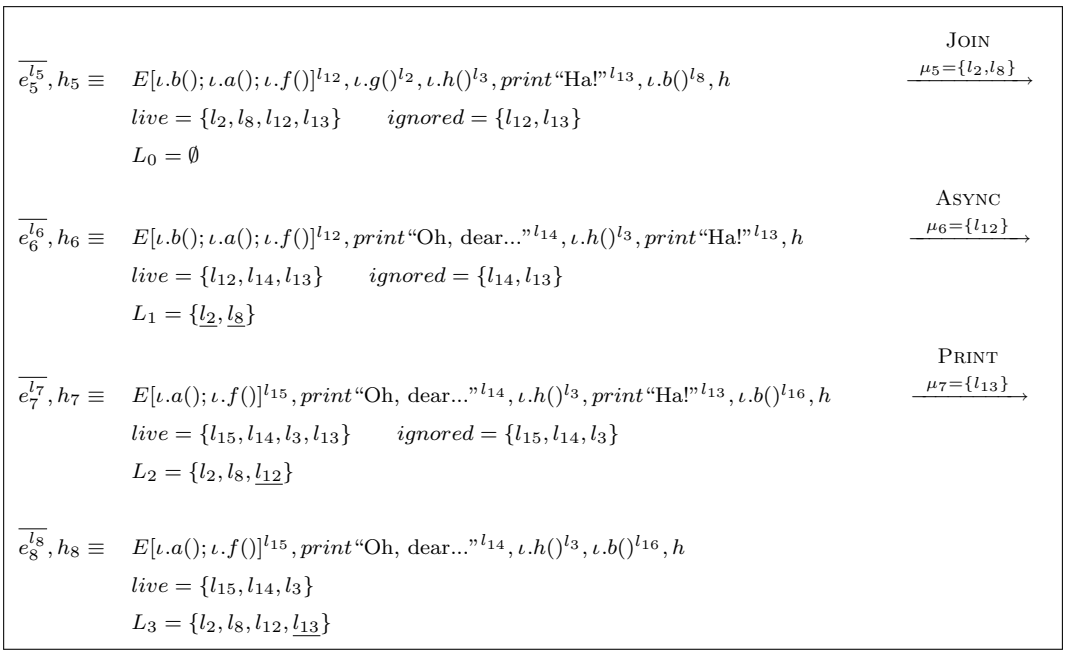

Figure 4.12: Next locally weakly-fair execution to be concatenated.

through 8 also is locally weakly-fair, written $\overline{e_5^{l_5}}, h_5 \overset{M'}{\underset{L'}{\Longrightarrow}} \overline{e_8^{l_8}}, h_8$, where $M' = \{l_2, l_8\} \circ \{l_{12}\} \circ \{l_{13}\}$ and $L' = \{l_2, l_8, l_{12}, l_{13}\}$. The concatenation of these two locally weakly-fair sequences results in a weakly-fair sequence of length 2, written $\overline{e_0^{l_0}}, h_0 \Rightarrow \overline{e_8^{l_8}}, h_8$. Indeed, no label has remained live throughout the execution sequence, and the message "Ha!" was printed as expected; the message "Oh, dear..." will eventually print if we continue execution with the next locally weakly-fair sequence, however, the message "Help!", as expected, may never print.

4.4.4 Correctness

In order to show correctness (theorem 10 below) for weakly-fair executions, we first establish correctness (lemma 17 below) 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; then, we assume a label remains always live and use the lemma to reach a contradiction.

Lemma 17 (Correctness of Locally Weakly-Fair Execution)

$$\left.\begin{array}{l} \overline{e^l}, h \stackrel{\substack{M \\ \Longrightarrow \\ L}}{} \overline{e'^{l'}}, h' \quad \text{where} \\ \overline{e^l}, h \equiv \overline{e_0^{l_0}}, h_0 \quad \text{and} \quad \overline{e'^{l'}}, h' \equiv \overline{e_k^{l_k}}, h_k \quad \text{for some} \quad k \in \mathbb{N} \quad \text{and} \\ \overline{e_0^{l_0}}, h_0, L_0 \stackrel{\mu_0}{\Longrightarrow} \overline{e_1^{l_1}}, h_1, L_1 \stackrel{\mu_1}{\Longrightarrow} \ \ldots \ \stackrel{\mu_{k-1}}{\Longrightarrow} \overline{e_k^{l_k}}, h_k, L_k \end{array}\right\} \Longrightarrow$$

$$\forall\, l \in live\left(\overline{e_0^{l_0}}\right) \ : \ \exists\, k', 0 < k' \leqslant k \ : \ l \notin live\left(\overline{e_{k'}^{l_{k'}}}\right)$$

Proof *By repeated application of the* WEAK *selection rule which corresponds to the locally weakly-fair execution; since by definition 26 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 10 (Correctness of Weakly-Fair Execution)

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

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

Proof *By contradiction, using lemma 17: assume a label l from configuration i becomes live at configuration $j \geqslant i$ and remains live $\forall\ k \geqslant j$ configurations; but then l will be live at some configuration $k' \geqslant j$ which is also the initial configuration of a locally weakly-fair execution sequence of some length γ, hence by the above lemma $\exists\ k'', k' < k'' \leqslant k' + \gamma - 1$: $l \notin live\left(\overline{e_{k''}^{l_{k''}}}\right)$.* □

Figure 4.13 contains a visual description of the above theorem; grey circles indicate configurations at which the label l is live, and the rectangle indicates a locally weakly-fair execution sequence.

4.4.5 Correspondence

A weakly-fair execution sequence corresponds to a sequence of locally weakly-fair execution sequences, which in turn corresponds to an ^{l}SCHOOL execution sequence (theorem 11 below). Then, from theorem 9 of section 4.2.2 above and this correspondence result we can obtain weakly-fair SCHOOL executions.

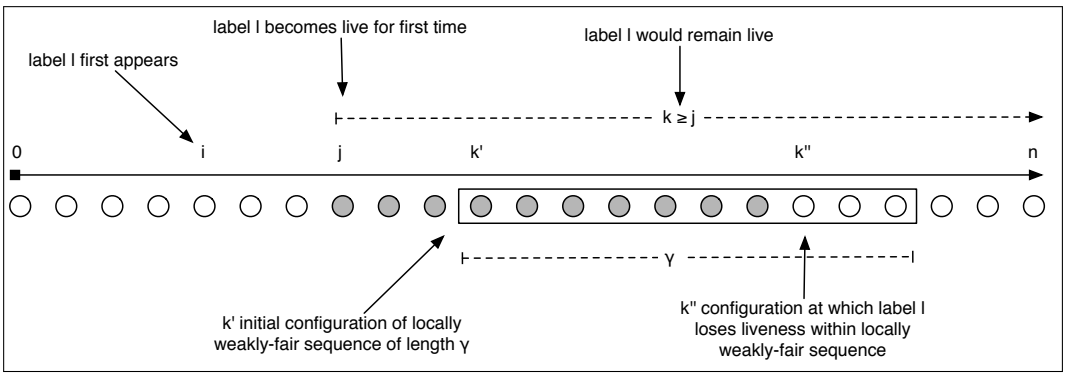

Figure 4.13: Visual description of correctness theorem for weak fairness.

Theorem 11 (Weakly-Fair and ^{l}SCHOOL Correspondence)

$$\overline{e^{l}}, h \;\Rrightarrow\; \overline{e'^{l'}}, h' \Longrightarrow$$

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

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.* □

4.5 Strong Fairness

We describe a mechanism which, with an appropriate implementation, realises the strong fairness constraints for ^{l}SCHOOL. We show how the original example from section 4.3 of an execution inadmissible under strong fairness is altered by imposing the constraints so as to become admissible. Furthermore, we present worst-case calculations for the liveness behaviour of labels under strong fairness.

Like with weak fairness, some aspects of the mechanism remain unspecified, resulting in the semantics being non-deterministic with regards to choice of labels and evaluation rules. Again, it is up to the implementation of a strongly-fair scheduler to specify these selection functions.

4.5.1 Overview

An execution sequence is strongly-fair when no label becomes live infinitely often; accordingly, starting from any given configuration, each of the live labels appearing in that configuration must eventually lose its liveness forever. Since executions can have an infinite length, it is not possible to first generate all valid executions and then select those which are strongly-fair; hence, we must employ a mechanism which *maintains* a strongly-fair execution as it is being generated.

While initially we are free to begin an execution with the selection of any applicable evaluation rule, the result of its application imposes several constraints on future selections. This is true for each application of an evaluation rule, and thus constraints accumulate throughout. Therefore, our mechanism must keep track of these constraints; in order to do so we introduce a *queue of labels*, which is modified at each evaluation step and passed to the next. The queue records the order of appearance of labels, and enables a current selection to take into consideration previous ones. All labels are included in the queue, resulting in a notion of completeness for the recording procedure: essentially, the queue is a constrained permutation of all the active labels in the current configuration.

Strong fairness is maintained by imposing certain constraints through a selection rule: 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 (any further participating labels can be at any location within the queue).

For a live label to be ignored it must be preceded by another live label in the queue which was selected for participation and subsequently removed. 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.

The strong fairness constraints may *force* the selection of a label (and consequently may force the selection of an evaluation rule which is applied to the label). Therefore, although the order in which labels are placed into the queue and the selection of applicable evaluation rule remain unspecified, strongly-fair executions may reach a configuration where the next evaluation step is completely determined by the constraints.

4.5.2 Strongly-Fair Execution

We define a strongly-fair execution (definition 29 below) as the execution sequence which consists of strongly-fair evaluation steps (definition 28 below), which are obtained through the STRONG selection rule of figure 4.14.

Definition 28 (Strongly-Fair Evaluation Step)
A strongly-fair evaluation step has the following form:

$$e_1^{l_1}, \ldots, e_n^{l_n}, h, Q \stackrel{\mu}{\Longrightarrow} e_1^{'\,l'_1}, \ldots, e_m^{'\,l'_m}, h', Q'$$

where an initial labelled configuration, $e_1^{l_1}, \ldots, e_n^{l_n}, h$*, evaluates to a final labelled configuration,* $e_1^{'\,l'_1}, \ldots, e_m^{'\,l'_m}, h'$*, 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* $m \geqslant n$*. We abbreviate a strongly-fair evaluation step as:* $\overline{e^l}, h, Q \stackrel{\mu}{\Longrightarrow} \overline{e'^{l'}}, h', Q'$*. A (possibly infinite) sequence of strongly-fair evaluation steps thus has the form:*

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

$$\frac{\begin{array}{l}\overline{e^l}, h \xrightarrow{\mu} \overline{e'^{l'}}, h' \\ \forall\, l \in live\left(\overline{e^l}\right) \;:\; \exists\, l \in live\left(\overline{e^l}\right) \;:\; l \leqslant_Q l \;\wedge\; l \in \mu \\ Q' = Q \upharpoonright active\left(\overline{e'^{l'}}\right) \circ active\left(\overline{e'^{l'}}\right) \setminus active\left(\overline{e^l}\right)\end{array}}{\overline{e^l}, h, Q \overset{\mu}{\Longrightarrow} \overline{e'^{l'}}, h', Q'}\ \textsc{Strong}$$

Figure 4.14: Strongly-fair selection rule.

A queue, Q, consists of a finite sequence of labels, denoted $\langle l_1 \circ \ldots \circ l_k \rangle$ for k labels, and its size, denoted $|Q|$, is k. An empty queue is denoted $\langle\rangle$. The leftmost label is the head and the remainder is the tail. The relation $l \leqslant_Q l'$ 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.

Two operations are defined on queues: removing labels from the queue, and appending labels to the queue; neither operation affects the order of the labels. Removing labels from a queue invovles the *retain* operator, denoted $\upharpoonright$, which is applied to a queue Q and a set of labels S as $Q \upharpoonright S$, and results in a new queue, Q', where all those labels in Q which are not in S have been removed, and the remainder labels have not changed order. For instance, for $Q = \langle l_1 \circ l_2 \circ l_3 \circ l_4 \rangle$ and $S = \{l_3, l_5, l_1\}$, we have $Q \upharpoonright S = \langle l_1 \circ l_3 \rangle$.

Appending a queue of labels $\langle l_{k+1} \circ \ldots \circ l_{k+h} \rangle$ to Q, written as $Q \circ \langle l_{k+1} \circ \ldots \circ l_{k+h} \rangle$, results in a new queue, Q', where the appended labels appear rightmost and the existing labels have not changed order: $Q' = \langle l_1 \circ \ldots \circ l_k \circ l_{k+1} \circ \ldots \circ l_{k+h} \rangle$. We require the appending operator, $\circ$, to have lower priority from other set-based operators. 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. Once the labels have been appended as a queue, however, their order does not change. For instance, appending the set of labels $\{l_c, l_b, l_a\}$ to Q *may* result in the queue $\langle l_1 \circ \ldots \circ l_k \circ l_b \circ l_a \circ l_c \rangle$.

The premises of STRONG require an applicable evaluation rule, ensure the first live label in the queue participates in the evaluation, only labels which remain active in the resulting configuration are retained in the queue (and thus all consumed labels are removed from the queue), and all newly created labels are appended to the queue.

In order to establish a notion of completeness for the recording of strong fairness constraints (i.e. manipulation of the queue), the definition of strongly-fair executions below requires a finite initial configuration and an initial queue which contains all initially active labels.

Definition 29 (Strongly-Fair Execution)

$\overline{e^l}, h, Q \;\Rrightarrow\; \overline{e'^{l'}}, h', Q'$ iff

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

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

$$\text{where} \quad \overline{e^l}, h, Q \equiv \overline{e_0^{l_0}}, h_0, Q_0 \quad \text{and} \quad \overline{e'^{l'}}, h', Q' \equiv \overline{e_n^{l_n}}, h_n, Q_n$$

$$\wedge \quad \exists\, m \in \mathbb{N} \;:\; |labels\left(\overline{e_0^{l_0}}\right)| = m$$

$$\wedge \quad Q_0 = \langle\rangle \circ active\left(\overline{e_0{}^{l_0}}\right)$$

4.5.3 Example

We consider an example of a strongly-fair execution in figure 4.15, similar to the inadmissible execution of figure 4.9. Again, we will use the class `StrongFairnessExample` from listing 4.4, and the same initial configuration (indexed by 0) which consists of two synchronous invocations of methods `f` and `g` respectively, and an asynchronous invocation of method `a`. New labels added to the queue are underlined, and the first live label in the queue is denoted with a hat.

The first four evaluation steps coincide with the original example: the first step involves the evaluation rule JOIN and labels l_1 and l_3 participating through the joining of the first chord, and hence l_2 being ignored; the second,

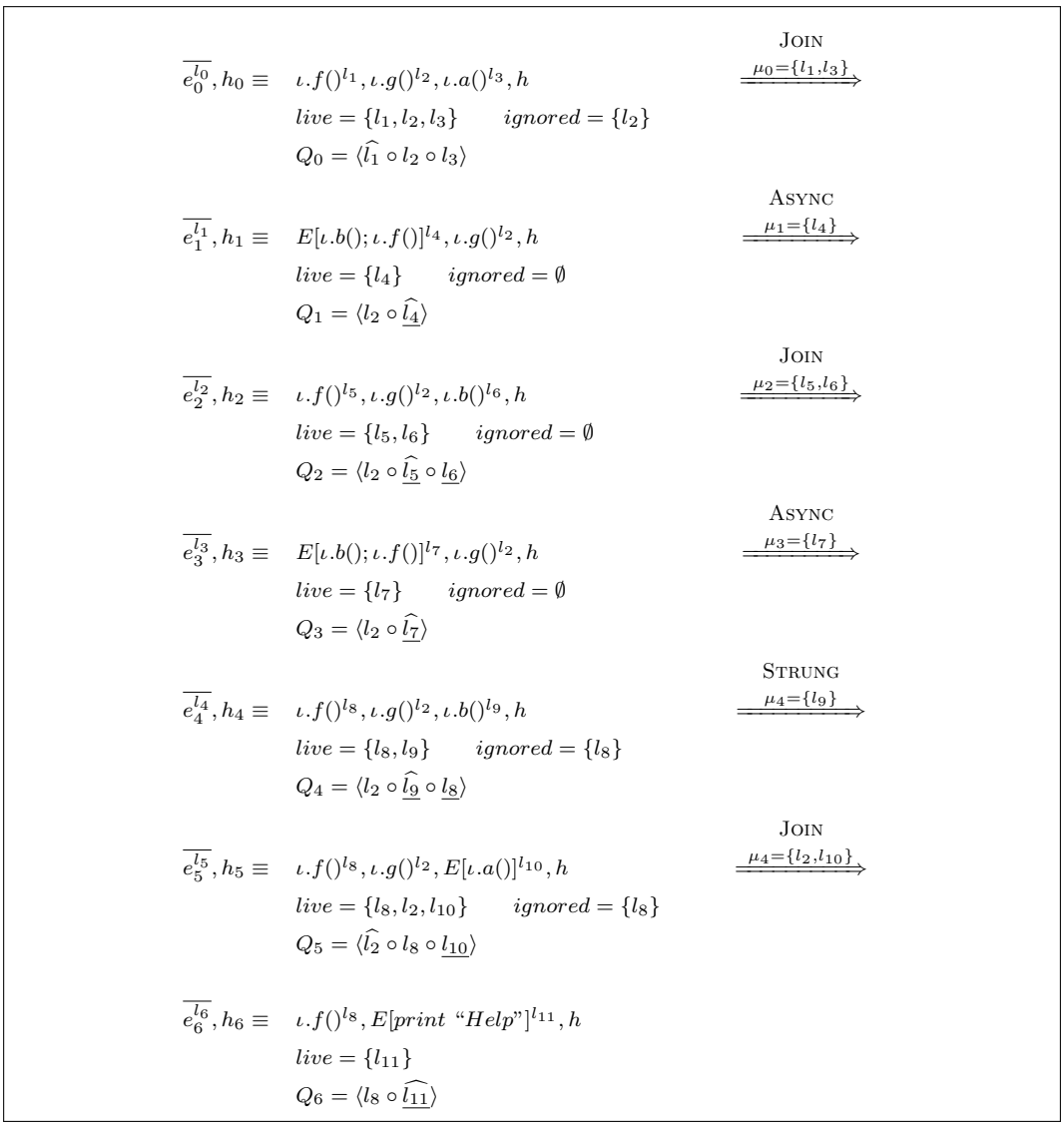

Figure 4.15: A strongly-fair execution.

third and fourth steps involved respectively the evaluation rules ASYNC, JOIN and ASYNC once more.

At this point in the original execution 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; therefore, since no label would be ignored infinitely often, the execution would be admissible as strongly fair.

However, if the execution ever involved the evaluation rule STRUNG with the current asynchronous invocation of method `b`, then l_2 would regain its liveness in the resulting configuration. If we were to allow l_2 to be ignored for the second time (as we did 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 inadmissible under strong fairness.

Therefore, let us assume that after the fourth evaluation, the label l_9 is placed before the label l_8 in the queue Q_4. Then 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. At this point, 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, thus participating in the next evaluation step.

Given a finite initial configuration, all further configurations in an execution are finite, and consequently all queues are finite. Therefore, although a label can be ignored several times, the actual number of times is finite, as the label moves closer to the head of the queue by at least one place each time it is ignored, and is never preempted. It is not, however, possible to determine how many times a label will be ignored as this is a direct consequence of its

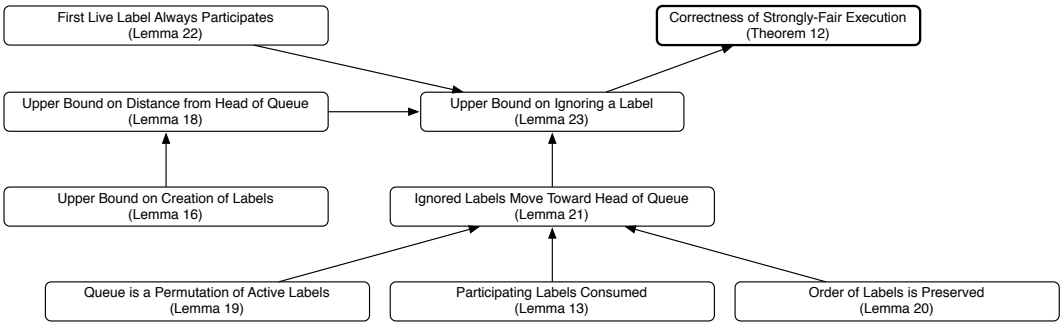

Figure 4.16: Overview of correctness theorem and lemmas.

placement in the queue and its relative order to other competing labels, which is unspecified.

4.5.4 Correctness

In order to show correctness (theorem 12 below) for the strongly-fair execution we first establish the existence of an upper bound on the number of times each label appearing in the execution can be ignored (lemma 23 below). Using this upper bound it is straightforward to see that, after a finite number of configurations from the configuration at which a label appears for the first time, the label will lose its liveness forever, either by participating in the evaluation or by never regaining its liveness. In order to establish the upper bound we first show that for each label there is an upper bound on its distance from the head of the queue (lemma 18 below), that each time a label is ignored it moves toward the head of the queue (lemma 21 below), and finally that the first live label always participates in the next evaluation step (lemma 22 below).

In order to establish the upper bound on the distance of each label from the head of the queue we need the result which establishes an upper bound on the number of labels which can be created at each evaluation step (lemma 16 from section 4.2.2 above). In order to establish that ignored labels move

toward the head of the queue we need three results: that the queue associated with a configuration is a permutation of the active labels of the configuration (lemma 19 below), that the order of labels in the queue is preserved (lemma 20 below), and finally that participating labels are consumed and do not appear in resulting configurations (lemma 13 from section 4.2.2 above).

Figure 4.16 provides an overview of the correctness theorem and lemmas of this section.

Lemma 18 (Upper Bound On Distance From Head Of Queue)
$\overline{e^l}, h, Q \;\Rrightarrow\; \overline{e'^{l'}}, h', Q' \Longrightarrow$

$$\forall\, i \in \mathbb{N} \;:\; l \in Q_i \Longrightarrow$$
$$\exists\, k, \lambda, \rho \in \mathbb{N}, k < i * \lambda + \rho \;:\; Q_i = \langle l_1 \circ \ldots \circ l_k \circ l \circ \ldots \rangle$$

Proof *By induction on the length of the execution and use of lemma 16.* □

Lemma 19 (Queue Is A Permutation Of Configuration)
$\overline{e^l}, h, Q \;\Rrightarrow\; \overline{e'^{l'}}, h', Q' \;\Longrightarrow\; \forall\, i \in \mathbb{N} \;:\; Q_i$ is a permutation of *active* $\left(\overline{e_i^{l_i}}\right)$

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

Lemma 20 (Order of Labels Preserved)
$\overline{e^l}, h, Q \;\Rrightarrow\; \overline{e'^{l'}}, h', Q' \Longrightarrow$

$$\forall\, i \in \mathbb{N} \;:\; \forall\, l, l' \in Q_i \;:\; l \leqslant_{Q_i} l' \;\wedge\; l, l' \in Q_{i+1} \;\Longrightarrow\; l \leqslant_{Q_{i+1}} l'$$
$$\wedge\;\; \forall\, l \in Q_i \;:\; \forall\, l' \in Q_{i+1}, l' \notin Q_i \;:\; l \leqslant_{Q_{i+1}} l'$$

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 21 (Ignored Labels Move Toward Head Of Queue)

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

Proof *Through application of the* STRONG *selection rule and using lemmas 13, 19 and 20.* □

Lemma 22 (First Live Label Participates)

$$\left.\begin{array}{l} \overline{e^{l}}, h, Q \;\Rrightarrow\; \overline{e'^{l'}}, h', Q' \\ Q_i = \langle l \circ \ldots \rangle \\ l \in live\left(\overline{e_i^{l_i}}, h_i\right) \end{array}\right\} \implies l \in \mu_i$$

Proof *Through application of the* STRONG *selection rule and the definition for the queue ordering operator* $\leqslant_Q$. □

Lemma 23 (Upper Bound On Ignoring A Label)

$\overline{e^{l}}, h, Q \;\Rrightarrow\; \overline{e'^{l'}}, h', Q' \implies$

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

Proof *Assume that a label l appears for the first time at configuration i, and hence by lemma 19 will be in the queue Q_i; then by lemma 18 we know that there is an upper bound on the distance of that label from the head of the queue, or:* $\exists\; k', \lambda, \rho \in \mathbb{N}, k' < i * \lambda + \rho \;:\; Q_i = \langle l_1 \circ \ldots \circ l_{k'} \circ l \circ \ldots \rangle$. *From lemma 21 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:* $Q_{i+t} = \langle l \circ \ldots \rangle$. *From lemma 22 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, $\{\tau_1, \ldots, \tau_\kappa\}$, for t, and hence:* l ignored at $i + t$ for some $t \in \mathbb{N} \implies t \in \{\tau_1, \ldots, \tau_k\}$. □

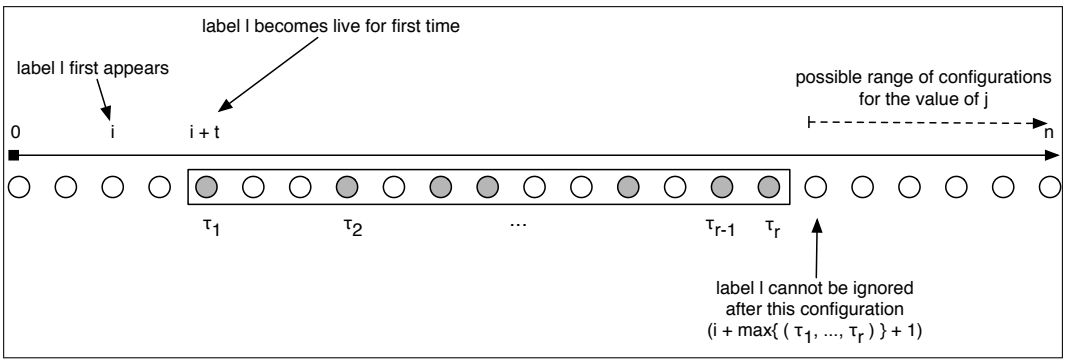

Figure 4.17: Visual description of correctness theorem for strong fairness.

Theorem 12 (Correctness of Strongly-Fair Execution)
$\overline{e^l}, h, Q \;\Rightarrow\; \overline{e'^{l'}}, h', Q' \Longrightarrow$
$\forall\, i \in \mathbb{N} \;:\; \forall\, l \in labels\left(\overline{e_i^{l_i}}\right) \;:\; \exists\, j \geqslant i \;:\; \forall\, k \geqslant j \;:\; l \notin live\left(\overline{e_k^{l_k}}\right)$

Proof *From lemma 23 we know that each label, l, will be ignored at configurations $i + t$ for some $t \in \{\tau_1, \ldots, \tau_r\}$ and finite r; but then we immediately obtain that $\exists\, j \geqslant i + max(\{\tau_1, \ldots, \tau_r\}) + 1 \;:\; \forall\, k \geqslant j \;:\; l \notin live\left(\overline{e_k^{l_k}}\right)$.*
□

Figure 4.17 contains a visual description of the above theorem; grey circles indicate configurations at which the label l is live, and the rectangle indicates the finite sequence of configurations from the initial ignore to the final ignore of the label.

4.5.5 Correspondence

A strongly-fair execution sequence corresponds to an ^{l}SCHOOL execution sequence (theorem 13 below). Then, from theorem 9 of section 4.2.2 above and this correspondence result we can obtain strongly-fair SCHOOL executions.

Theorem 13 (Strongly-Fair and ^{l}SCHOOL Correspondence)
$\overline{e^l}, h, Q \;\Rightarrow\; \overline{e'^{l'}}, h', Q' \Longrightarrow$
$\overline{e_0^{l_0}}, h_0 \xrightarrow{\mu_0} \overline{e_1^{l_1}}, h_1 \xrightarrow{\mu_1} \overline{e_2^{l_2}}, h_2 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_{n-1}} \overline{e_n^{l_n}}, h_n \xrightarrow{\mu_n} \ldots$

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

4.5.6 Worst-Case Liveness Behaviour

From observation 1 we know that, in the absence of fairness constraints, a label can either alternate between live and non-live always, or alternate α times and then remain always live or remain always non-live. We assume that a label, l, appears for the first time in the i'th configuration. Then, depending on the liveness behaviour of the label, we show that in both cases strong fairness ensures that there is an upper bound on the number of times l will be ignored.

In order to calculate the upper bound on the number of configurations after which l will lose its liveness forever we use the result from lemma 23 that l will be ignored at most $((i * \lambda) + \rho - 1)$ times. In order to obtain the worst case, we maximise the number of times a label can be ignored and the number of configurations before the last configuration at which it is ignored as per figure 4.7 and the results from table 4.1 of section 4.2.4.

If l would alternate always then strong fairness will force its participation in the worst case after the upper bound on the number of configuration at which l can be ignored is reached; the number of configurations at which this occurs depends on whether l starts are live or non-live.

If l would alternate α times then we must consider whether α would be odd or even in conjunction with whether l starts as live or non-live. If l starts live then an even α means l would at some point remain always live, while an odd α means l would at some point remain always non-live; the converse also holds.

Furthermore, we must consider whether the number of alternations would lead to l being ignored more than the upper bound on the number of times a label can be ignored under strong fairness; hence, we must consider whether $\lceil \alpha \rceil \leqslant ((i * \lambda) + \rho - 1)$ in conjunction, again, with whether l starts as live or

Livenss behaviour of l	Value of j such that $\forall\ k \geqslant j\ :\ l \notin live\left(\overline{e_k^{l_k}}\right)$
Alternates always, and	
l starts non-live:	$i + (\theta + 1) * ((i * \lambda) + \rho - 1) + \theta + 1$
l starts live:	$i + (\theta + 1) * ((i * \lambda) + \rho - 1) + 1$
Alternates α times, and	
α is odd, and	
l starts non-live, and	
$\lceil \alpha/2 \rceil \leqslant ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * \lceil \alpha/2 \rceil + ((i * \lambda) + \rho - 1) - \lceil \alpha/2 \rceil + 1$
$\lceil \alpha/2 \rceil > ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * ((i * \lambda) + \rho - 1) + \theta + 1$
l starts live, and	
$\lceil \alpha/2 \rceil \leqslant ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * \lfloor \alpha/2 \rfloor + 1$
$\lceil \alpha/2 \rceil > ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * ((i * \lambda) + \rho - 1) + 1$
α is even, and	
l starts non-live, and	
$(\alpha/2) \leqslant ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * (\alpha/2)$
$(\alpha/2) > ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * ((i * \lambda) + \rho - 1)$
l starts live, and	
$(\alpha/2) \leqslant ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * (\alpha/2) + ((i * \lambda) + \rho - 1) - (\alpha/2) + 1$
$(\alpha/2) > ((i * \lambda) + \rho - 1)$:	$i + (\theta + 1) * ((i * \lambda) + \rho - 1) + 1$

Table 4.2: Summary of worst-case calculations for loss of liveness depending on liveness behaviour and initial conditions.

non-live, and whether α is even or odd.

Table 4.2 summarises the calculations for the worst case of liveness behaviour of a label under strong fairness, depending on initial conditions (whether α would be odd or even, less than or greater than the upper bound of times a label can ignored, and whether the label starts as live or non-live).

Figure 4.18 illustrates two cases of worst-case liveness behaviour under strong fairness (since we want the worst case we maximise the number of steps between configurations at which the label l is ignored by having each sequence of configurations in which it is non-live have length θ). White circles are configurations at which the label is non-live, grey circles are configurations at which the label is live, and black circles are configurations at which the label *would have been* live if strong fairness constraints were not imposed.

In the top execution the liveness behaviour of l would be to alternate always,

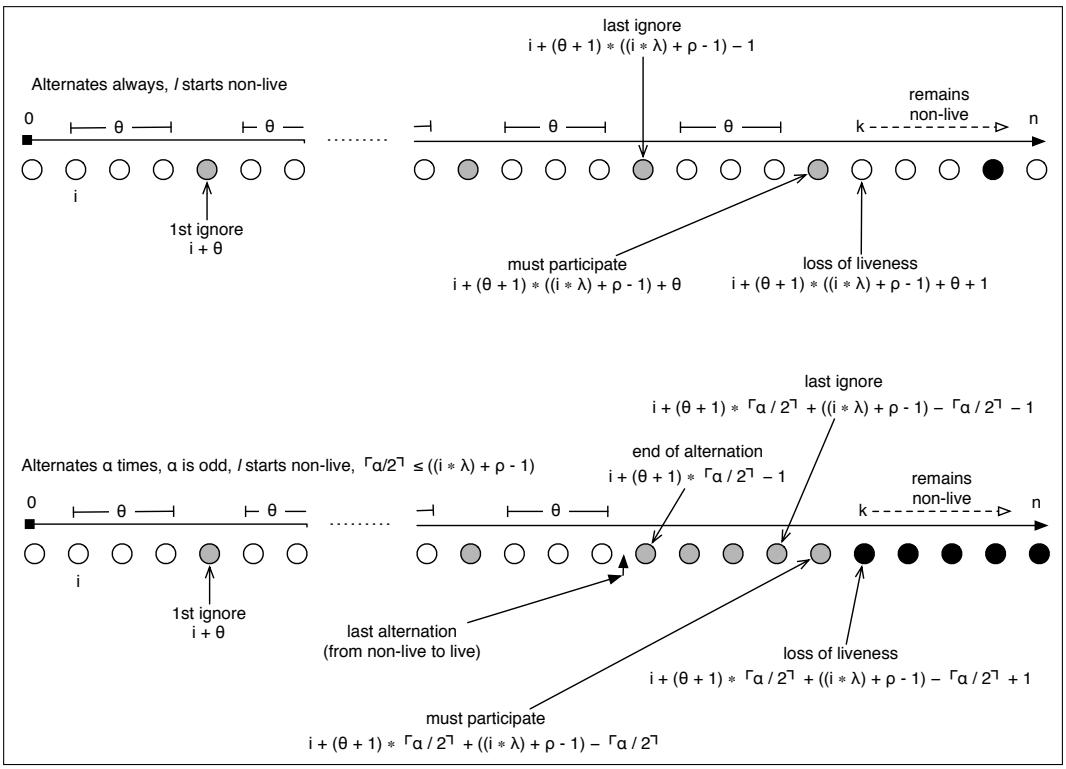

Figure 4.18: Two examples of worst-case liveness behaviour under strong fairness.

and so the strongly-fair selection rules force the participation of the label after it reaches the upper bound on the number of times a label can be ignored.

In the bottom execution l would alternate α times, where α would be odd and l start out as non-live (and hence l would remain live after its last alternation). It happens that l would remain always live starting at a configuration before the upper bound on the number of configurations at which it can be ignored is reached, so the strongly-fair selection rules allow it to be ignored for a finite number of consecutive configurations before forcing it to participate.

4.6 Summary and Conclusions

In this chapter we presented a labelling mechanism for SCHOOL, resulting in ^{l}SCHOOL, and defined the properties of fresh labels, liveness, and ignoring a

label; this enabled us to state weak and strong fairness in terms of SCHOOL executions. We then presented weakly-fair and strongly-fair abstract schedulers. Finally, we identified a notion of alternating liveness behaviour of labels which enabled us to show worst-case delays for processes under strong fairness.

In general, weak fairness results in an abstract scheduler which gives greater freedom of implementation than the one for strong fairness, as there is no specified mechanism by which locally weakly-fair execution sequences are generated. Although this increases the number of valid schedulers one could employ in an implementation of a chorded language, it also increases the uncertainty in reasoning about the execution of programs. On the other hand, a strongly-fair scheduler may be overly restrictive for some classes of programs, as there can be very little, local deviation from the enforced queueing. It remains a matter of future investigation on the kinds of programs people wish to write using chords before an informed choice between fairness notions can be made.

In the concluding chapter we will briefly overview how chords instead of processes can be scheduled, and a straightforward way of modifying the labelling of ^{l}SCHOOL so that liveness properties are stated in terms of chords and the selection rules for both weakly- and strongly-fair schedulers remain the same. We will also explore the possibility of scheduling process and chords simultaneously, and point out several problems which arise and make the description of such schedulers significantly more complex.

Chapter 5

Conclusions

5.1 Summary of Thesis

We started our work with an explicit interest in the design and implementation of programming languages used for complex concurrent systems, as such systems are becoming more widely used. The various problems which arise from programming with concurrency have been partially addressed in contemporary object-oriented programming languages through explicit concurrency constructs such as threads, locks and monitors. However, it is not clear that such constructs have assisted in the correct understanding and development of concurrent software. We therefore explored an alternative concurrency paradigm based on the *chemical semantics* of the language Γ which was later formalised through the Join Calculus and introduced into object-oriented languages with the name of *chords* in Polyphonic C♯.

We noticed that many implementations of chords have been proposed, such as JoCaml, Scala, C_ω, Funnel, and others; however, although chords were inspired by a small formal calculus, there was no such model for any chorded language. We believe a formalisation of chords and other language constructs which are to be used alongside will prove beneficial in the understanding of chorded programs' behaviours.

We proposed a featherweight model for chorded languages, SCHOOL, which consists of a generalised definition of a chord, a small term rewriting system which describes chorded semantics, and a nominal single-inheritance type system. Using this model we attempted to capture the essence of chord behaviours and obtained formal definitions for the fundamental properties of type safety and progress.

We then extended SCHOOL to include mutable fields, a common construct of object-oriented languages, resulting in ^{f}SCHOOL; this extension had two goals: first, through a translation from ^{f}SCHOOL into SCHOOL we showed that mutable fields are not a necessary construct for chorded languages as they can be encoded using only chords, and hence do not increase the expressive power of the language; second, we presented a methodology which we hope can be used to study other common language constructs and their interaction with chords, such as explicit thread life-cycle control and re-entrant monitors.

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. Many programming examples from these languages require some notion of fairness in order to work or be useful (for instance rendezvous or readers/writers). Although there is a long and solid understanding of several fundamental notions of fairness, namely weak and strong, there is no study of such notions as they apply to chorded languages.

We therefore chose to define weak and strong fairness for SCHOOL, and devised a labelling mechanism (^{l}SCHOOL) which allows us to create abstract schedulers for chorded languages which satisfy the two notions of fairness. Furthermore, the labelling mechanism gives us a basis for stating the aforementioned fairness notions, as well as properties such as liveness, liveness behaviours (such as worst-case and alternating liveness), and freshness of labels.

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.

Finally, 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.

5.2 Suggestions for Future Work

We suggest four ideas for future work based on the material presented so far. First, we suggest alternative derivations of the general chord construct (section 5.2.1 below) in order to obtain either a more general concept or a more expressive one; then we suggest some fundamental issues which arise when exceptions are introduced into the language (section 5.2.2 below); furthermore we give an overview of some problems which arise once explicit concurrency constructs are included in a chorded language, namely explicit thread life-cycles and re-entrant monitors (section 5.2.3 below); finally, we investigate the scheduling of chords rather than processes, as well as a combination of the two, and identify some of the problems which arise when notions of fairness

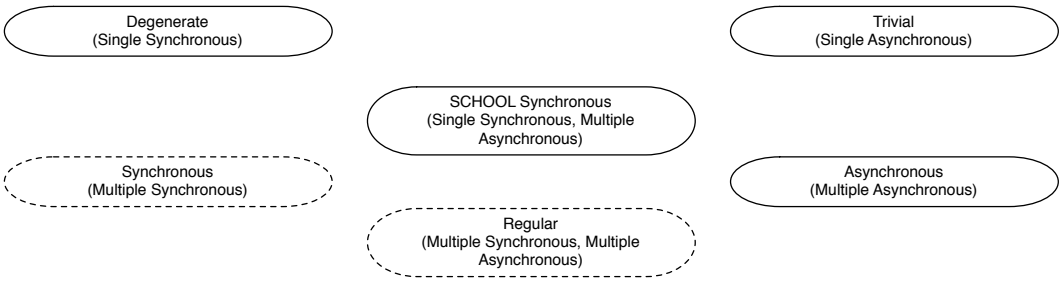

Figure 5.1: Chord Derivation Asymmetry in SCHOOL

are presented in the context of combined scheduling (section 5.2.4 below).

5.2.1 Alternative Derivation of Chords

The derivation of a generalised chord from section 3.2 consists of a combination of synchronous and asynchronous methods in the chord header. This combination results in four kinds of chords: normal (one synchronous and one or more asynchronous methods), asynchronous (two or more asynchronous methods), degenerate (a single synchronous method), and trivial (a single asynchronous method). Furthermore, the possibility of *void* methods to participate as both synchronous and asynchronous parts of a chord header results in non-linear join patterns, which affects progress (section 3.3.4 above).

Polyphonic C$^\sharp$ features both synchronous and asynchronous methods, as it makes sense from an implementation viewpoint: a continuous execution of a synchronous chain of method invocations can be easily realised with a single thread. Similarly, JoCaml features synchronous channels (through the *reply/to* construct) which can be mixed with asynchronous channels in join patterns in order to facilitate a functional programming paradigm.

As synchronous methods (or channels) can be encoded with continuations, they are not a necessary construct in a chorded language. Furthermore, although their presence facilitates a functional aspect of a chorded language, the

presence of synchronous methods presents an irregularity in the join patterns of SCHOOL: there are no chords which feature multiple synchronous methods (either on their own or in combination with multiple asynchronous methods); figure 5.1 depicts the lack of symmetry in SCHOOL (dashed outlines represent chords which cannot be defined).

There are two ways in which a general derivation of chords can become regular: either include multiple synchronous methods in a chord header, or remove synchronous methods altogether. The latter would result in a smaller language (the JOIN rule would not be necessary), although such a language would be further away from existing chorded languages. If such a change is made, then one would have to present an encoding of synchronous methods from SCHOOL into the smaller language using continuations, and show that the encoded programs are equivalent, much like the presentation of field encoding in section 3.4.

The former would result in a larger language because some additional mechanism would be necessary in order to execute synchronous code which is attached to multiple synchronous methods. In SCHOOL, the body of a chord is executed in the evaluation context of the process which invoked the synchronous method of that chord; when multiple processes are blocked on a single chord, it is unclear where the body of the chord should evaluate. One solution is to spawn a new special process which executes the body, and provide some mechanism of forwarding the return values to each of the participating synchronous methods. This could be realised by creating a counterpart evaluation rule to JOIN, say COMPLETE, which collects the return values of the above special processes and places them in each of the relevant evaluation contexts of the invoker processes.

By choosing to remove synchronous methods from SCHOOL, the language will feature strictly linear join patterns, a property which is shared with the join calculus. If, however, synchronous methods are permitted, then linearity

can only be achieved if *void* method cannot participate in both parts of a chord header.

5.2.2 Exceptions and Chords

Exceptions are a common feature of many contemporary object-oriented languages, and are available in many of the chorded languages, such as Polyphonic C♯ and Scala, as well as part of the languages used for library-based implementation of chords, such as the *Joins Library* (.NET, VB, and so on) or *Join Java*. Therefore, it would be useful to investigate some common models of exceptions in the context of chorded languages.

Typically, when an exception occurs (is raised, or thrown) either control flow is redirected to a previous evaluation context, to a predefined handler (usually a body of code), or the exception is termed *uncaught* and is handled by the runtime system, or a default handler specified by the user. Although some aspects of exceptions can be easily incorporated into SCHOOL (such as redirecting control to a previous evaluation context), several important issues arise which require careful consideration. In the remainder of this section we give brief overviews of such issues.

Exception Definitions

A fundamental aspect of any exception model for chorded languages would be selecting which entities can be defined as throwing exceptions: methods, or chords, or both. By having methods throw exceptions, the language will be closer to the semantics of most contemporary object-oriented languages. By having chords throw exceptions, the emphasis is placed on the chords, which may be more desirable as a programming paradigm.

Depending on who is responsible for handling exceptions (see next section), either approach may be desirable. One model would have methods throwing

exceptions, and by consequence a chord throws all exceptions thrown by each of its participating methods. Another model would have chords throwing exceptions, and thus each participating method throws all exceptions thrown by each of the chords in which it participates. Finally, a combined model would have both methods and chords throwing exceptions, and both of the aforementioned rules would apply. The last model is probably overly complex, and the second can become complicated once inheritance is introduced, as methods may acquire new exceptions by participating in new chords defined in sub-classes.

Exception Locations and Handler Matching

It is possible to define the *location* of an exception occurrence as either being within an evaluation context of a particular process (the process which invoked the synchronous method of a chord, or the new process which was created in order to execute a strung chord), or within the body of a chord. In the former case, it is the responsibility of the invoking code to define an exception handler (somewhere along the invocation chain). In the latter case, the exception handler is tied to the particular chord, and will be used irrespective of the invoking process.

The benefit of the former approach is that the same exception can be treated differently by different invokers; the benefits of the latter approach are that asynchronous chords (with no invoker) can specify exception handlers, and that chords with multiple synchronous methods (from section 5.2.1 above) can be accommodated, as there is no need to decide which one among multiple threads should handle the exception.

A combined approach could also be employed, and precedence rules could result in models where, for instance, an exception which is not caught by the (an) invoking process is sent to the chord handler, rather than up the invoke chain. In the light of asynchronous chords, it is probably desirable for chord-

specific exception handlers to be a feature of the language.

Ground Values, Null Pointers and Deadlock

Progress in SCHOOL (theorem 2 of section 3.3.4 above) must consider null pointers (invocations on null values) when deciding whether a configuration can continue execution or is deadlocked (or terminated). Because the language does not have a model for exceptions, such null pointers are treated in an ad-hoc manner, and are equivalent to ground values when considering progress. A model for exceptions would hence allow for a more systematic treatment of blocked configurations due to uncaught exceptions.

Uncaught exceptions could be treated as ground values, and hence contribute towards termination (when all values are ground); alternatively, they could be considered errors, and a configuration with one or more uncaught exceptions could be considered as not having terminated. Indeed, as with section 3.3.4 above, the semantics which one wishes to ascribe to termination can be elaborated by incorporating an exception model into the language.

Semantics of The Async Type

In SCHOOL, a *void* return type of a method enables the method to participate as both a synchronous and an asynchronous part of a chord. This is possible because a method of return type *void* can be implemented to return immediately, as the invoker is not expecting a return value. In the event, however, of a method being defined as potentially throwing an exception, the exception definition becomes part of the type signature of that method. In such a situation, the invoker is responsible for handling the exception, and hence the invoker must be able to wait for the method to return (successfully or to throw an exception).

Obviously then, it is not possible to define asynchronous methods which throw exceptions, unless default handlers are defined for such situations (han-

dlers which are independent from the state of the method's invoker process). This has two important ramifications for SCHOOL: first, it is no longer possible for *void* methods to participate asynchronously (unless a special case of a *void* method which does not throw exceptions is defined, and inheritance does not break the condition); second, in order to define asynchronous methods a special type *async* would need to be introduced as a sub-type of *void*.

5.2.3 Explicit Concurrency

We have seen from chapter 2 that many chorded languages also feature explicit concurrency constructs typical in object-oriented languages, such as thread control and monitors. This characteristic is not due to a known benefit of mixing chords with other concurrency semantics, rather, it is because chords have been added to existing language which already feature rich concurrency semantics.

A systematic study of the interaction between chords and other concurrency constructs would benefit our understanding of existing chorded languages, as well as offer guidance in the design of future languages. Below we will briefly look at the kinds of issues which arise when chords interact with thread life-cycles and monitors.

A fundamental question which one must answer when considering the combination of chords with explicit concurrency constructs is whether such constructs should be (automatically) encoded using only chords, when possible, similar to the encoding of fields from section 3.4. As we have seen from section 2.2.2, JoCaml has the capacity to encode all synchronous behaviours with continuations, however, the designers of the language chose to include synchronous execution in the language; although this did not increase the number of constructs which can be expressed in JoCaml, it was implemented as a consideration for the programmer.

Explicit Thread Life-Cycles

Explicit control of threads typically involves a mechanism by which threads can be referred to, usually by unique identifiers, and possibly a mechanism of organising threads into groups, which may also be hierarchical. Depending on the characteristics of the threading model, threads can be observed, spawned, suspended, resumed, terminated, and waited upon; in the presence of thread groups there may be explicit control on the allocation of threads from a pool, or the addition and removal of threads from a pool.

Extending SCHOOL with explicit thread control constructs would at the very least necessitate the introduction of syntax and a mechanism for naming threads and referring to them through identifiers; furthermore, the definition of progress (and hence deadlock) would have to be extended to take into consideration the reason a thread cannot execute. For instance, one could decide that a thread which is explicitly waiting for another thread is blocked, while a thread which has been explicitly suspended is equivalent to a terminated expression. Such design decisions, as with our own decision on top-level asynchronous method invocations from section 3.3.4, will be guided by the intended use of threads with chords rather than some intrinsic property of their semantics.

A further consideration is whether some threads may be anonymous; specifically, threads which are implicitly spawned in order to execute the bodies of asynchronous chords are not created by an expression, and hence are not named by the user. In such situations it is possible to provide a facility which automatically names threads and allows code to discover their name. However, such a mechanism may overly expose the implementation: code will be able to determine whether it is running as a result of an asynchronous chord joining, a property of the language which may be undesirable in the interest of encapsulation and information hiding.

Another issue is whether *void* methods should continue to be able to participate in the asynchronous parts of chords: given that the bodies of *void* methods may be executing in threads which become suspended explicitly, it is quite likely that the intent of the user is to have the invoker of the method wait as well. Unlike the similar issue with *void* methods and exceptions above (section 5.2.2), whether such methods should be only synchronous in the presence of explicit thread life-cycles is a decision related to programming.

Re-entrant Monitors

Monitors are typically used to guarantee mutually exclusive access to a resource by multiple competing processes. A monitor consists of a queue of waiting threads and a mechanism by which threads can register their interest in *acquiring* the monitor (they "enter the queue" and wait) and a notification procedure which "wakes up" a thread from the queue which next acquires the monitor. It is possible for a single thread to be (non-deterministically) selected, or all threads to be notified via a "broadcast wake" and compete for the monitor anew. A re-entrant monitor keeps track of which thread has currently acquired it and allows multiple acquisitions from the same thread to pass through in the case of nested acquisition requests.

Although re-entrant monitors require some mechanism by which threads are identified, this mechanism need not be exposed to the user, as in the case of explicit thread life-cycles above. Indeed, the labelling mechanism of ^{l}SCHOOL is sufficient for an implementation of monitors as it uniquely identifies each thread. Therefore, it should be straightforward to include two rules, say Enter and Acquire, which model the entering of a process into the monitor's queue and the acquisition of the monitor respectively; since manipulation of the monitor may include a "broadcast" notification which wakes all waiting threads in the monitor's queue, these rules will be of the many-to-many re-write form.

Monitors would benefit from explicit thread life-cycle constructs, as then it becomes possible to issue an "interrupt" request to the monitor (or thread which has acquired it) and arbitrarily release the monitor and all its waiting threads.

In order to implement monitors in SCHOOL a construct which contains the monitors becomes necessary. One option is to use objects as monitors, as is done in the Java language for example. An explicit queue is not a necessary addition to objects, since the non-deterministic choice of which thread "wakes up" allows an implementation similar to our implementation of method invocation queues: an evaluation context with the term *acquire(ι)* for instance suffices to identify the process as waiting to acquire the monitor represented by the object at address ι. However, in order to implement re-entrant behaviour, the current thread which has acquired the monitor must be stored somewhere, and this could be implemented as a special label field of objects.

5.2.4 Alternative Scheduling

It is straightforward to suggest the development of other well-known notions of fairness, such as those briefly mentioned in section 2.3.8; however, here we will not deal with an incremental repertoire of abstract schedulers, rather, we will suggest a more fundamental change to scheduling which may yield greater flexibility in writing programs.

The abstract schedulers presented in chapter 4 deal with processes; in this respect all liveness properties are stated only in terms of processes, rather than chords, or a combination of the two. It is possible to devise schedulers which deal with chords instead of processes, or a combination of chords and processes, so that liveness properties can be stated for chords as well. In this section we overview some issues which may arise.

Scheduling Chords

Scheduling chords instead of processes may be desirable if one suggests that in a chorded language the most important construct is the chord, and hence its constituent body of code; since program behaviour is organised around chords, they have precedence over processes when considering scheduling.

With the schedulers presented in chapter 4 the mechanism for labelling and referring to individual processes was straightforward, involving an annotation of each process with a unique label. A mechanism for referring to chords, however, is more involved, primarily because a chord remains unchanged in the program definition independently of whether its body is executed. Furthermore, there is no obvious "location" in which to store annotations, such as along with each process. Therefore, a special construct must be employed which assigns a unique label to each chord; additionally, these labels must be able to change in order to establish a notion of freshness.

Once chords can be referred to, it is straightforward to state the existing weak and strong schedulers in such terms. Indeed, once appropriate modifications to the label look-up functions *labels*, *active* and *live* are in place, the selection rules Weak and Strong remain unchanged. The labelled operational semantics, on the other hand, need to be modified so as to move the labelling mechanism into the new construct. One straightforward way of doing this would be to include a chord name-to-label mapping Λ where chord names are canonical representations of chords generated by concatenating all their participating methods. For instance, $\Lambda(fab) = \lambda_3$ would designate that the chord with header `int f()` & `async b()` & `async a()` is currently labelled with λ_3; then labelled execution has the form: $\overline{e}, h, \Lambda \xrightarrow{\mu} \overline{e'}, h', \Lambda'$.

An important consideration when scheduling chords is how to treat newly-introduced chords due to inheritance: if a subclass modifies a chord, should both chords be recorded in Λ separately, or should all refinements of a chord

map to the same entry? Recording refinements due to inheritance is technically straightforward, as the canonical names of chords can simply be prepended with the name of the class they are defined in. However, whether this is desirable scheduling semantics will depend on how programmers aim to use inheritance in a chorded language.

Another issue of note is that under strong fairness, once a chord reaches the head of the queue it *must* be selected; however, where with processes the first live process was forced into the selection but the remainder processes were selected non-deterministically, with chords there are no remainder ones to be selected. This may result in an overly constraining scheduler, and perhaps scheduling chords is more suitable for weakly-fair executions.

Multidimensional Scheduling

It may be desirable to include the selection of which chord, in addition to which process, in the scheduler. This mechanism would reflect (an equal) importance of both chords and processes in chorded programs. As we have seen in the section above, applying the selection rules from chapter 4 to chords instead of process is straightforward; however, devising selection rules which consider both simultaneously turns out to raise several important issues.

One possible solution for scheduling chords is to create a two-tier scheduler, where the primary selection involves chords, and the secondary selection involves processes. This will require the existing mechanism of schedulers (a serviced set for weak fairness and a queue for strong fairness), as well as the mechanism mentioned in the section above with a chord labelling location Λ; therefore, the primary serviced set or queue consists of labels from Λ, while a secondary serviced set or queue consists of process labels as per ^{l}SCHOOL.

The selection procedure then first constrains, based on the primary serviced set or queue, and once a chord is selected, the secondary serviced set or queue is consulted; the liveness of a chord and the liveness of processes which

currently consist of a method invocation to the chord's participating methods will consistently match, so there is no possibility of selecting a chord which cannot join - the inverse would not be true, as the totality of live processes is not necessarily sufficient to join each chord.

One way of keeping track of which chords and processes are live is via a chord-label-to-process-label-set Ψ: for instance, $\lambda_4 \mapsto \{l_3, l_9, l_{12}\}$ signifies that the chord with label λ_4 is live, and a proper subset of the processes with labels in the set $\{l_3, l_9, l_{12}\}$ can be selected in order for the chord to join. To make this example concrete, consider the program fragment:

```
1 void f() & async a() { print ‘‘First’’; }
2 async a() { print ‘‘Second’’; }
3 void g() & async b() { print ‘‘Third’’; }
```

where the chords are respectively labelled with λ_1, λ_2, and λ_3, and the current configuration is $\iota.f()^{l_1}, \iota.g()^{l_2}, \iota.a()^{l_3}, \iota.a()^{l_4}, h$. Thus, Λ may consist of the following ordering of chord labels: $\langle \lambda_3, \lambda_1, \lambda_2 \rangle$, while Q may consist of the following ordering of process labels: $\langle l_4, l_1, l_2, l_3 \rangle$. Hence, Ψ contains the mappings $\{\lambda_1 \mapsto \{l_1, l_3, l_4\}, \lambda_2 \mapsto \{l_3, l_4\}\}$. Under strong fairness, for instance, the first live chord from Λ is selected, which is λ_1; the first live process label in Q from the set $\{l_1, l_3, l_4\}$ which maps to λ_1 is selected, which is l_4; finally, the only other process label which can be selected is l_1, resulting in the joining of the first chord and chord λ_2 and process l_3 being ignored.

The above mechanism, however, does not directly deal with individual live expressions within chord bodies, as these expressions are not about to join. It is not clear what selection process will adequately solve this problem, since the execution of individual expressions and the joining of chords must be merged in a way which satisfies the desired fairness notion.

One promising direction would be to include two-dimensional scheduling, where the serviced set or queue has one dimension for chords and another for processes. Using a two-dimensional serviced set for weakly-fair scheduling

would result in a system where a locally weakly-fair execution is established once a chord and process label combination from each row and column are selected. For strongly-fair scheduling, the "head" of the queue would be the origin, and labels would move "up" and "left" towards the head.

It is clear that the established abstract schedulers cannot be extended to treat chords (or objects) simultaneously with processes in a straightforward way; a two-tier scheduler will probably be simpler to describe but require some arbitrary decisions on how to merge chord joins with individual expression evaluation, while a two-dimensional scheduler will probably result in a more regular model but may be overly complex to implement or be useful for reasoning about the execution of chorded programs.

5.3 Closing Remarks

It is encouraging that during our work on SCHOOL several implementations of chords have appeared for various languages, such as Concurrent Basic [79], an extension of Visual Basic, and the chord co-ordination framework for C$^\sharp$ [19]. Both language extensions are derived primarily from the work on Polyphonic C$^\sharp$ and C$_\omega$; these recent developments suggest that chords continue to interest the programming language community and our work remains relevant.

We feel that the featherweight model for chords which we have developed captures a fundamental subset of all chorded languages and will prove useful for the future study of chords. As a consequence, the results yielded by our investigation of the principal notions of weak and strong fairness for chorded languages will remain relevant for such future study of chorded languages and their associated constructs.

It seems that weak fairness, in particular, is a suitable basis for the design of schedulers for chorded languages: the spirit of the original chemical metaphor and the Join Calculus requires as few bottlenecks and as little centralisation as

possible, ideally being realised as a truly concurrent system, something weak fairness coincides with. In contrast to weak fairness, strong fairness necessitates a queueing mechanism which forms a bottleneck and hence slightly diverts from the spirit of the original metaphor.

We hope that future work will result in notions of fairness for chorded languages which will guarantee useful liveness properties while staying as close as possible to the original metaphor.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York Inc., 1998.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[3] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. In *Journal of Functional Programming*, volume 7, pages 1 – 72. Cambridge University Press, New York, NY, USA, 1997.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. In *Information Processing Letters*, volume 21, pages 181–185. Elsevier, 1985.

[5] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[6] Krzysztof R. Apt and E. R. Olderog. Proof rules and transformations dealing with fairness. *Science of Computer Programming*, 3:65–100, 1983.

[7] Krzysztof R. Apt and E. R. Olderog. Transformations realizing fairness assumptions for parallel programs. In *Proceedings of the Symposium of Theoretical Aspects of Computer Science*, volume 166 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, London, UK, 1984.

[8] J.W. Backus, F.L. Bauer, J.Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language algol 60. Technical report, ACM Committee on Programming Languages and the GAMM Committee on Programming, 1959.

[9] H.P. Barendregt. The lambda calculus. In *Studies in Logic and the Foundations of Mathematics*, number 103. Elsevier, 1997.

[10] Nelson H. F. Beebe. Bibliography on the fortran programming language. Technical report, University of Utah, 2008.

[11] Jean-Pierre Benâtre, Anne Coutant, and Daniel Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.

[12] Jean-Pierre Benâtre and Daniel Le Metayer. A new computatinal model and its discipline of programming. Technical Report INRIA 566, 1986.

[13] Jean-Pierre Benâtre and Daniel Le Metayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, 1990.

[14] Nick Benton, Gavin Bierman, Luca Cardelli, Erik Meijer, Claudio Russo, and Wolfram Schulte. C_ω. `http://research.microsoft.com/Comega/`, 2004.

[15] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C♯. In B. Magnusson, editor, *Proc. of ECOOP02*, volume 2374, pages 415–440. Springer Press, 2002.

[16] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C♯. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[17] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94. ACM Press, 1990.

[18] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[19] Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for C♯. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL) Workshop, OOPSLA*, October 2005.

[20] E. M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. *Annual Review of Computer Science*, 2:269–290, 1987.

[21] Gerardo Costa and Colin Stirling. Weak and strong fairness in ccs. *Information and Computation*, 73(3):207–244, 1987.

[22] Vincent Cremet. Join definitions in scala. `http://lamp.epfl.ch/cremet/join_in_scala/`.

[23] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall Inc., 1976.

[24] Sophia Drossopoulou, Alexis Petrounias, Alex Buckley, and Susan Eisenbach. SCHOOL: a small chorded object-oriented language. In Maribel Fernández and Ian Machie, editors, *Proceedings of ICALP '05 First International Workshop on Developments in Computational Models*, pages 55–64, 2005.

[25] Fabrice Le Fessant. The jocaml language (older, unmaintained edition). `http://moscova.inria.fr/oldjocaml/`, 2008.

[26] Cédric Fournet. The jocaml language. `http://jocaml.inria.fr/`, 2008.

[27] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alain Schmitt. Jocaml: a language for concurrent distributed and mobile programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming: 4th International School, AFP 2002*, volume 2638 of *LNCS*. Springer-Verlag, 2003.

[28] 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.

[29] Cédric Fournet and Georges Gonthier. The join calculus: a language for distributed mobile programming. *Applied Semantics Summer School*, pages 1–66, 2008.

[30] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes In Computer Science*, pages 406 – 421, 1996.

[31] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *Proc. of the 1997 8th International Conference on Concurrency Theory*. Springer-Verlag, 1997.

[32] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join-calculus (extended abstract). In *Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *LNCS*, pages 397–408. Springer-Verlag, December 2000.

[33] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join-calculus (technical report). In *Journal of Logic and Algebraic Programming*, 2004.

[34] Nissim Francez. *Fairness*. ACM Press, New York, USA, 1986.

[35] Dov Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM Conference on Principles of Programming Languages, Las Vegas*, 1980.

[36] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[37] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Welsey, 2006.

[38] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[39] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[40] D. Le Métayer J.-P. Banâtre. Gamma and the chemical reaction model: Ten years after. In *Coordination programming: mechanisms, models and semantics*, 1996.

[41] Simon Payton Jones. Haskell 98 language and libraries. Technical report, haskell.org, 2002.

[42] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:269–272, 1994.

[43] R. Kuiper and W. P. de Roever. Fairness assumptions for csp in a temporal logic framework. In Dines Bjorner, editor, *Proceedings of the 2nd IFIP Working Conference on Formal Description of Programming Concepts*, pages 159–170, 1983.

[44] M. Z. Kwiatkowska. *Fairness for non-interleaving concurrency.* PhD thesis, University of Leicester, United Kingdom, 1989.

[45] M. Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989.

[46] Leslie Lamport. Proving the correctness properties of multiprocess programs. In *IEEE Transactions in Software Engineering*, volume 3, pages 125–143, 1977.

[47] Leslie Lamport and Fred B. Schneider. *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190, chapter Formal Foundation for Specification and Verification, pages 203–285. Springer-Verlag, London, UK, 1985.

[48] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(157), 1966.

[49] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. *Automata, Languages and Programming*, 115:264–277, 1981.

[50] Qin Ma. Prototyping on object-oriented join calculus. Technical report, DEA Programmation, Université Paris 7, 2001.

[51] Qin Ma. *Concurrent Classes and Pattern Matching in the Join Calculus.* PhD thesis, INRIA-Rocquencourt and Université Paris 7, 2005.

[52] Qin Ma and Luc Maranget. Expressive synchronization types for inheritance in the join calculus. In *Proc the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *LNCS*. Springer-Verlag, 2003.

[53] Qin Ma and Luc Maranget. Compiling pattern-matching in join-patterns. In *Proc. of the 15th International Conference on Concurrency Theory*, volume 3170 of *LNCS*. Springer-Verlag, 2004.

[54] Zohar Manna and Amir Pnueli. Verification of concurrent programs: The temporal framework. *The Correctness Problem in Computer Science*, pages 215–273, 1981.

[55] Luc Maranget and Fabrice Le Fessant. Compiling join-patterns. In *Electronic Notes Theoretical Computer Science*, volume 16, 1998.

[56] G.J. Milne and R. Milner. Concurrent processes and their syntax. *Journal of the ACM*, 26(2):302–321, 1979.

[57] R. Milner. An approach to the semantics of parallel programs. In *Proceedings Convegno di Informatica Teoretica*, pages 285–301, 1973.

[58] R. Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings Logic Colloquium*, number 80 in Studies in Logic and the Foundations of Mathematics, pages 157–174. North-Holland, 1975.

[59] R. Milner. Algebras for communicating systems. In *Proceedings AFCET/SMF joint colloquium in Applied Mathematics*, 1978.

[60] R. Milner. Synthesis of communicating behaviour. In J. Winkowski, editor, *7th MFCS*, number 64, pages 71–83. Springer-Verlag, 1978.

[61] R. Milner. Flowgraphs and flow algebras. *Journal of the ACM*, 26(4):794–818, 1979.

[62] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, (92), 1980.

[63] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, (100):1–77, 1992.

[64] Robin Milner. A finite delay operator in synchronous css. Technical Report CSR116 -82, University of Edinburgh, 1982.

[65] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[66] Demetrios Nicolaou. A virtual machine for a chorded programming language. Master's thesis, Department of Computing, Imperia College London, 2007.

[67] Martin Oderksy. An overview of functional nets. Technical report, APPSEM Summer School, 2000.

[68] Martin Oderksy, Christoph Zenger, and Matthias Zenger. A functional view of join. Technical Report CRC-99-016, University of South Australia, 1999.

[69] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS. Springer Verlag, 2000.

[70] David Park. On the semantics of fair parallelism. *Abstract Software Specifications*, 86:504–526, 1980.

[71] David Park. Concurrency and automata on infinite sequences. *Theoretical Computer Science*, 104:167–183, 1981.

[72] David Park. A predicate transformer for weak fair iteration. In *Proceedings of the 6th IBM Symposium on Mathematical Foundations of Computer Science, Hakone, Japan*, 1981.

[73] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[74] Gordon D. Plotkin. A powerdomain for countable non-determinism (extended abstract). *Automata, Languages and Programming*, 140:418–428, 1982.

[75] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. *Current Trends in Concurrency*, 224:510–584, 1986.

[76] J. P. Queille and J. Sifakis. Fairness and related properties in transition systems — a temporal logic to deal with fairness. *Acta Informatica*, 19(3):195–220, 1983.

[77] J. H. Reppy. Cml: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN, Conference on Programming Language Designand and Implementation*, pages 293–305, 1991.

[78] Claudio Russo. The joins concurrency library. In *PADL*, 2007.

[79] Claudio Russo. Join patterns for visual basic. In *OOPSLA*, 2008.

[80] Fred B. Schneider. Decomposing properties into safety and liveness. Technical Report TR87-874, Cornell University, Ithaca, NY, USA, 1987.

[81] G. Smolka, M. Henz, , and J. Wàurtz. *Principles and Practice of Constraint Programming*, chapter 2: Object-oriented concurrent constraint programming in Oz, pages 29–48. MIT Press, 1991.

[82] M. B. Smyth. Power domains and predicate transformers: A topological view. In *Automata, Languages and Programming*, volume 154 of *Lecture*

Notes in Computer Science, pages 662–175. Springer Berlin / Heidelberg, 1983.

[83] D. A. Turner. Sasl language manual. Technical report, University of Kent, 1983.

[84] D. A. Turner. An overview of miranda. In *SIGPLAN Notices*, volume 21, pages 158–166, 1986.

[85] Daniele Varacca and Hagen Völzer. New perspectives on fairness. In *Concurrency column - Bulletin of the EATCS*, 2006.

[86] Lefteris Volanakis. Implementation of a chorded compiler. Master's thesis, Department of Computing, Imperial College London, 2006.

[87] Hagen Völzer, Daniele Varacca, and Ekkart Kindler. Defining fairness. Technical Report SIM-TR-A-05-18, University of Lübeck, 2005.

[88] G. Stewart von Itzsein. *Introduction of High Level Concurrency Semantics in Object Oriented Languages*. PhD thesis, University of South Australia, December 2005.

[89] G. Stewart von Itzsein and Mark Jasiunas. On implementing high level concurrency in java. In *Advances in Computer Systems Architecture*, 2003.

[90] G Stewart von Itzstein and David Kearney. Applications of join java. In *CRPITS '02: Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, pages 37–46. Australian Computer Society, Inc., 2002.

[91] Tim Wood. A chorded compiler for java. Master's thesis, Imperial College, London, June 2004.