A Featherweight Model for Chorded Languages

Alexis Petrounias Sophia Drossopoulou Susan Eisenbach

January 15, 2009

Abstract

Chords are a concurrency mechanism of object-oriented languages inspired by the join of the Join-Calculus. 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.

1 Introduction

The *chord* construct is a concurrency mechanism inspired by the *join* from the Join-Calculus [13]. Chords were implemented in Polyphonic C$^\sharp$ [5], an extension of C$^\sharp$, and are also available in C$_\omega$ [3]. Their proponents say that their use will raise the level of abstraction concurrent programs are written in, and make the development of correct programs easier.

Furthermore, the inclusion of chords in C$_\omega$ means they will be used in conjunction with imperative concurrency constructs such as monitors and threads. In order to study the interactions between other language constructs and chords, it is necessary that we fully understand the behaviour of chords themselves.

We therefore provide 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).

SCHOOL turned out to be surprisingly small, featuring just four rules and permutation. Yet, the model can describe a rich set of chorded behaviours, including generalisations of those found in current experimental languages such as Polyphonic C$^\sharp$ or Scala with *join* [9].

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 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. Proofs are available from: `slurp.doc.ic.ac.uk/chords/school/`. Our work has evolved out of a substantially more complicated model [11], called SCHOOL$_0$ in this paper, which we have worked on previously.

The structure of this paper is as follows: section 2 provides an overview of chords and programming with them; section 3 presents the formalisation of SCHOOL, several of its properties including soundness and progress, and a comparison with the older model for SCHOOL and with Polyphonic C$^\sharp$; section 4 presents ^{f}SCHOOL and its properties, the encoding of ^{f}SCHOOL in SCHOOL, and the proof of equivalence between SCHOOL and encoded ^{f}SCHOOL programs; section 6 overviews related work; and section 7 concludes.

2 Overview 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 1) with a traditional implementation (in Java, listing 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 `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`).

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

Listing 1: Chorded Countdown Latch

```
class CountdownLatch {
 int permits;
 void  await(int permits) throws InterruptedException {
  synchronized(this) {
   this.permits = permits;
   while (permits > 0) { wait(); }
  }
 }
 void countDown() {
  synchronized(this) {
   if (--permits == 0) { notify(); }
  }
 }
}
```

Listing 2: Java Countdown Latch

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 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$^\sharp$ suffer from similar problems.

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$^\sharp$, 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 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.

3.1 Abstract Syntax and Program Representation

Figure 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.

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: $Id^c \rightarrow Id^c$, 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: $Id^c \times Id^m \rightarrow MethSig$; 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: $t\ m(c)$.

Abstract Syntax

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

Program Representation

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

Runtime Entities

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

Figure 1: SCHOOL overview.

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

Figure 2: SCHOOL representation of the Countdown Latch program.

The third component maps a class name to the set of chords defined in the class: $Id^c \to \mathcal{P}(Chord)$; 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 1 is represented in abstract syntax as in figure 2, ignoring the rule about number of parameters, and assuming that arithmetic and integers are defined in the language.

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:

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

Evaluation Contexts

$$E[\cdot] \quad ::= \quad [\cdot] \;\mid\; E[\cdot].m(e) \;\mid\; \iota.m(E[\cdot]) \;\mid\; E[\cdot];e$$

Evaluation Rules

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

$$\frac{h(\iota){=}c \qquad m \in \bigcup_{\chi \in P\downarrow_3(c)} \chi\downarrow_2 \qquad E[\cdot] \neq [\cdot]}{E[\iota.m(v)], h \quad\longrightarrow\quad E[voidValue], \iota.m(v), h}\ \textsc{Async}$$

$$\frac{h(\iota){=}c \qquad (m, \{m_1, \ldots, m_k\}, e) \in P\downarrow_3(c)}{\begin{array}{l} E[\iota.m(v)], E_1[\iota.m_1(v_1)], \ldots, E_k[\iota.m_k(v_k)], h \;\longrightarrow \\ \quad E[e[{}^{\iota}/_{this}, {}^{v}/_{m\_x}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]], E_1[voidValue_1], \ldots, E_k[voidValue_k], h \end{array}}\ \textsc{Join}$$

$$\frac{h(\iota){=}c \qquad (\varepsilon, \{m_1, \ldots, m_k\}, e) \in P\downarrow_3(c)}{\begin{array}{l} E_1[\iota.m_1(v_1)], \ldots, E_k[\iota.m_k(v_k)], h \;\longrightarrow \\ \quad E_1[voidValue_1], \ldots, E_k[voidValue_k], E[e[{}^{\iota}/_{this}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]], h \end{array}}\ \textsc{Strung}$$

$$\frac{\overline{e} \;\cong\; \overline{e''e''''} \qquad \overline{e''}, h \;\longrightarrow\; \overline{e'''}, h' \qquad \overline{e'''e''''} \;\cong\; \overline{e'}}{\overline{e}, h \quad\longrightarrow\quad \overline{e'}, h'}\ \textsc{Perm}$$

Figure 3: SCHOOL operational semantics.

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:

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

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; 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 com-

position 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 $E[\centerdot] \neq [\centerdot]$, requiring the evaluation context to not be empty, is necessary so that we avoid infinite reductions of the form:

$$\begin{array}{l} \iota.m(v), h \quad \longrightarrow \quad \mathit{voidValue}, \iota.m(v), h \quad \longrightarrow \\ \quad \mathit{voidValue}, \mathit{voidValue}, \iota.m(v), h \quad \longrightarrow \quad \ldots \end{array}$$

- 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, 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 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 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 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 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

1		$\iota.await(2), \iota.countDown(null); \iota.countDown(null), h$
2	$\xrightarrow{\text{ASYNC}}$	$\iota.await(2), voidValue; \iota.countDown(null), \iota.countDown(null), h$
3	$\xrightarrow{\text{JOIN}}$	$(if\ 1 > 0)\ \iota.await(1), voidValue; \iota.countDown(null), voidValue, h$
4	$\xrightarrow{\text{SEQ}}$	$(if\ 1 > 0)\ \iota.await(1), \iota.countDown(null), voidValue, h$
5	$\xrightarrow{\text{IF}}$	$\iota.await(1), \iota.countDown(null), voidValue, h$
6	$\xrightarrow{\text{ASYNC}}$	$\iota.await(1), voidValue, voidValue, \iota.countDown(null), h$
7	$\xrightarrow{\text{JOIN}}$	$(if\ 0 > 0)\ \iota.await(0), voidValue, voidValue, voidValue, h$
8	$\xrightarrow{\text{IF}}$	$voidValue, voidValue, voidValue, voidValue, h$

Figure 4: Example of the Countdown Latch execution in SCHOOL.

Abstract Syntax

$$e \in Expr \quad ::= \quad \ldots \quad | \quad if\ (\ e\ )\ e$$

Evaluation Contexts

$$E[\bullet] \quad ::= \quad \ldots \quad | \quad if\ (\ E[\bullet]\ )\ e$$

Evaluation Rule

$$\frac{e' = \begin{cases} e & \text{if } v = true \\ voidValue & \text{otherwise} \end{cases}}{E[if\ (\ v\ )\ e], h \quad \longrightarrow \quad E[e'], h}\ \text{IF}$$

Figure 5: SCHOOL if-statements.

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.

3.3 Types, Well-Formedness and Subject Reduction

The type system of SCHOOL (presented in figure 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.

Type Definitions

$$\frac{P{\downarrow_1}(c) \text{ is defined}}{P \vdash_{\diamond_{cl}} c}\ \textsc{Def-Class} \qquad \frac{}{P \vdash_{\diamond_{cl}} \mathit{Object}}\ \textsc{Def-Class-Object}$$

$$\frac{P \vdash_{\diamond_{cl}} c}{P \vdash_{\diamond_{tp}} c}\ \textsc{Def-Type} \qquad \frac{}{P \vdash_{\diamond_{tp}} \mathit{void}}\ \textsc{Def-Type}$$

Typing Judgements

$$\frac{}{P, \Gamma, h \vdash \mathit{voidValue} : \mathit{void}}\ \text{T-}\textsc{Void} \qquad \frac{x \in \{\mathit{this}\} \cup \mathit{Id}^x}{P, \Gamma, h \vdash x : \Gamma(x)}\ \text{T-}\textsc{ThisX}$$

$$\frac{h(\iota){=}c}{P, \Gamma, h \vdash \iota : c}\ \text{T-}\textsc{Address} \qquad \frac{P \vdash_{\diamond_{cl}} c}{P, \Gamma, h \vdash \mathit{new}\ c : c}\ \text{T-}\textsc{New}$$

$$\frac{}{P, \Gamma, h \vdash \mathit{null} : c}\ \text{T-}\textsc{Null} \qquad \frac{\begin{array}{l} P, \Gamma, h \vdash e_1 : c \\ P, \Gamma, h \vdash e_2 : t \\ P{\downarrow_2}(c, m) = t_r\ m(t) \end{array}}{P, \Gamma, h \vdash e_1.m(e_2) : t_r}\ \text{T-}\textsc{Inv}$$

$$\frac{\begin{array}{l} P, \Gamma, h \vdash e_0 : \_ \\ P, \Gamma, h \vdash e : t \end{array}}{P, \Gamma, h \vdash e_0; e : t}\ \text{T-}\textsc{Seq}$$

Figure 6: SCHOOL type system.

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. $P{\downarrow_2}(c, m) = t\ m(\_) \implies \begin{cases} m \in \bigcup\limits_{\chi \in P{\downarrow_3}(c)} \{\chi{\downarrow_1}\} \cup \chi{\downarrow_2} & \text{if } t = \mathit{void} \\ m \in \bigcup\limits_{\chi \in P{\downarrow_3}(c)} \{\chi{\downarrow_1}\} & \text{otherwise} \end{cases}$
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 $\forall\ id \in \mathit{dom}(\Gamma)\ :\ P, \Gamma, h \vdash \sigma(id) : \Gamma(id)$

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 1 *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)
$e, h \quad \longrightarrow \quad e', h' \implies \forall\ \iota \in dom(h)\ :\ h(\iota) = h'(\iota)$

Proof 2 *By case analysis on the evaluation rule used in the derivation of* $e, h \quad \longrightarrow \quad e', h'$*. 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)

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

Proof 3 *By structural induction on the derivation of* $e, h \quad \longrightarrow \quad e', h'$*; 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)

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

Proof 4 *By structural induction on the derivation of* $e_1, \ldots, e_n, h \quad \longrightarrow \quad e'_1, \ldots, e'_n, e, h'$*; 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 subtype 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.

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.

3.4 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. $m \in \bigcup_{\chi \in P\downarrow_3(c)} \chi\downarrow_2 \implies E[\centerdot] = [\centerdot]$
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 $E[\centerdot] = [\centerdot]$).

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, $E[\centerdot] = [\centerdot]$, 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. $m \in \bigcup_{\chi \in P\downarrow_3(c)} \chi\downarrow_2 \implies E[\centerdot] = [\centerdot]$
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)

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

Proof 5 *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.*

3.5 Comparison Between SCHOOL and Polyphonic C$^\sharp$

There are several structural differences between Polyphonic C$^\sharp$ chords and SCHOOL chords:

- While Polyphonic C$^\sharp$ has methods and chords, we unify the two and treat Polyphonic C$^\sharp$ 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$^\sharp$ 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$^\sharp$ methods are either synchronous or asynchronous; SCHOOL does not have this restriction either.
- Polyphonic C$^\sharp$ class hierarchies are acyclic, while this is not a constraint imposed by SCHOOL.

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

3.6 Simplifications from Previous Model

Our original formulation of SCHOOL is presented in [11]; we will refer to it as SCHOOL$_0$. SCHOOL is substantially simpler than SCHOOL$_0$; we give a brief overview of the simplifications here.

In SCHOOL$_0$ method invocation is modelled as message passing between objects: the argument of the method being invoked is sent to the receiving object and subsequently placed within a queue corresponding to that method. Hence the structure of objects consists of their class name and a set of queues, one for each asynchronous method.

In the case of a SCHOOL$_0$ synchronous method call, waiting for a join to occur on a chord of a particular receiver object, it suffices to inspect only that object's queues. However, in the case of asynchronous chords there is no method call waiting for a join. Hence, it is necessary to inspect all objects whose class contains at least one asynchronous chord. Thus, detection of deadlock is a property of the expressions and the heap.

In SCHOOL deadlock can now be detected by inspecting only the set of executing expressions. This enables us to treat synchronous and asynchronous chords in a uniform way with respect to deadlock detection.

We have removed the special type *async*, using *void* as the return type of asynchronous methods. This makes the type system and proof of subject reduction simpler, as it dispenses with converting *async* to *void*.

Additionally, the use of the *void* return type for asynchronous methods enables us non-deterministically to chose whether such methods are invoked synchronously or asynchronously: all methods of return type *void* can participate in both the synchronous and asynchronous parts of chords.

We have also removed those constructs which can be programmatically encoded: null pointers, exceptions, and sequencing.

The semantics of SCHOOL is much simpler than that of SCHOOL$_0$: there are fewer rules (from nine versus five), and the rules are shorter.

3.7 Comparison Between SCHOOL and Polyphonic C$^\sharp$

There are several structural differences between Polyphonic C$^\sharp$ chords and SCHOOL chords:

- While Polyphonic C$^\sharp$ has methods and chords, we unify the two and treat Polyphonic C$^\sharp$ 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$^\sharp$ 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$^\sharp$ methods are either synchronous or asynchronous; SCHOOL does not have this restriction either.
- Polyphonic C$^\sharp$ class hierarchies are acyclic, while this is not a constraint imposed by SCHOOL.

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

Listing 3: Chorded Countdown Latch with fields.

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

3.8 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 [19] and Nicolaou [16] as part of final year undergraduate projects supervised by the author in the Department of Computing, Imperial College London, United Kingdom.

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 paper we use the annotation $^f\ldots$ to denote ^{f}SCHOOL entities.

Listing 3 shows the Chorded Countdown Latch example using fields. In the original example (listing 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 write to an object's fields using the dot operator.

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).

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 is represented as in figure 8; it is similar to the original representation from figure 2 of section 3.1, but now includes a fourth element for the field `permits`.

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, we use the symbol $\vdash_{\mathcal{f}}$ to disambiguate. We do not need to do the same for execution (figure 9) and types (figure 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:

Abstract Syntax

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

Program Representation

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

Runtime Entities

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

Figure 7: ^{f}SCHOOL overview.

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

Figure 8: ^{f}SCHOOL representation of the Countdown Latch program.

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:

Definition 8 (^{f}SCHOOL **Well-Formed Heaps)**
${}^{f}P \vdash_{\mathcal{F}} {}^{f}h$ 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 ${}^{f}P \vdash_{\mathcal{F}} {}^{f}h$.

Evaluation Contexts

$$E[\centerdot] \quad ::= \quad [\centerdot] \ \mid \ E[\centerdot].m({}^fe) \ \mid \ \iota.m(E[\centerdot]) \ \mid \ E[\centerdot];{}^fe \ \mid \ E[\centerdot].f \ \mid \ E[\centerdot].f = {}^fe \ \mid \ \iota.f = E[\centerdot]$$

Evaluation Rules

$$\frac{{}^fh(\iota) \text{ is undefined} \qquad {}^fP{\downarrow_4}(c) = \{f_1\ \_, \ldots, f_k\ \_\}}{E[new\ c], {}^fh \quad \longrightarrow \quad E[\iota], {}^fh[\iota \mapsto (c, \{f_1 \mapsto null, \ldots, f_k \mapsto null\})]}\ \textsc{New}$$

$$\frac{}{E[z;{}^fe], {}^fh \quad \longrightarrow \quad E[{}^fe], {}^fh}\ \textsc{Seq}$$

$$\frac{{}^fh(\iota){=}(c,\_) \qquad m \in \bigcup_{\chi \in {}^fP{\downarrow_3}(c)} \chi{\downarrow_2} \qquad E[\centerdot] \neq [\centerdot]}{E[\iota.m(v)], {}^fh \quad \longrightarrow \quad E[voidValue], \iota.m(v), {}^fh}\ \textsc{Async}$$

$$\frac{{}^fh(\iota){=}(c,\_) \qquad (m, \{m_1, \ldots, m_k\}, {}^fe) \in {}^fP{\downarrow_3}(c)}{\begin{array}{l} E[\iota.m(v)], E_1[\iota.m_1(v_1)], \ldots, E_k[\iota.m_k(v_k)], {}^fh \ \longrightarrow \\ \quad E[{}^fe[{}^{\iota}/_{this}, {}^{v}/_{m\_x}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]], E_1[voidValue_1], \ldots, E_k[voidValue_k], {}^fh \end{array}}\ \textsc{Join}$$

$$\frac{{}^fh(\iota){=}(c,\_) \qquad (\varepsilon, \{m_1, \ldots, m_k\}, {}^fe) \in {}^fP{\downarrow_3}(c)}{\begin{array}{l} E_1[\iota.m_1(v_1)], \ldots, E_k[\iota.m_k(v_k)], {}^fh \ \longrightarrow \\ \quad E_1[voidValue_1], \ldots, E_k[voidValue_k], E[{}^fe[{}^{\iota}/_{this}, {}^{v_1}/_{m_1\_x}, \ldots, {}^{v_k}/_{m_k\_x}]], {}^fh \end{array}}\ \textsc{Strung}$$

$$\frac{{}^fh(\iota){=}(c, \{\ldots, f \mapsto v, \ldots\})}{E[\iota.f], {}^fh \quad \longrightarrow \quad E[v], {}^fh}\ \textsc{Read}$$

$$\frac{{}^fh(\iota){=}(c, fs)}{E[\iota.f = v], {}^fh \quad \longrightarrow \quad E[v], {}^fh[\iota \mapsto (c, fs[f \mapsto v])]}\ \textsc{Write}$$

$$\frac{\overline{{}^fe} \ \cong \ \overline{{}^fe''{}^fe''''} \qquad \overline{{}^fe''}, {}^fh \quad \longrightarrow \quad \overline{{}^fe'''}, {}^fh' \qquad \overline{{}^fe'''{}^fe''''} \ \cong \ \overline{{}^fe'}}{\overline{{}^fe}, {}^fh \quad \longrightarrow \quad \overline{{}^fe'}, {}^fh'}\ \textsc{Perm}$$

Figure 9: ^{f}SCHOOL operational semantics.

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

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

Proof 6 *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* $\overline{{}^fe}, {}^fh \quad \longrightarrow \quad \overline{{}^fe'}, {}^fh'$*; we perform a case analysis of the last evaluation rule applied.*

Type Definitions

$$\frac{{}^{f}P{\downarrow_1}(c) \text{ is defined}}{{}^{f}P \ \vdash_{\diamond cl} \ c}\ \textsc{Def-Class} \qquad \frac{}{{}^{f}P \ \vdash_{\diamond cl} \ \mathit{Object}}\ \textsc{Def-Class-Object}$$

$$\frac{{}^{f}P \ \vdash_{\diamond cl} \ c}{{}^{f}P \ \vdash_{\diamond tp} \ c}\ \textsc{Def-Type} \qquad \frac{}{{}^{f}P \ \vdash_{\diamond tp} \ \mathit{void}}\ \textsc{Def-Type}$$

Typing Judgements

$$\frac{}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ \mathit{voidValue} : \mathit{void}}\ \text{T-}\textsc{Void} \qquad \frac{x \in \{\mathit{this}\} \cup \mathit{Id}^{x}}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ x : \Gamma(x)}\ \text{T-}\textsc{ThisX}$$

$$\frac{{}^{f}h(\iota){=}(c, \_)}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ \iota : c}\ \text{T-}\textsc{Address} \qquad \frac{{}^{f}P \ \vdash_{\diamond cl} \ c}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ \mathit{new}\ c : c}\ \text{T-}\textsc{New}$$

$$\frac{}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ \mathit{null} : c}\ \text{T-}\textsc{Null} \qquad \frac{\begin{array}{c} {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_1 : c \\ {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_2 : t \\ {}^{f}P{\downarrow_2}(c, m) = t_r \ m(t) \end{array}}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_1.m({}^{f}e_2) : t_r}\ \text{T-}\textsc{Inv}$$

$$\frac{\begin{array}{c} {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e : c \\ {}^{f}P{\downarrow_4}(c, f) = t \end{array}}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e.f : t}\ \text{T-}\textsc{Read} \qquad \frac{\begin{array}{c} {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_1 : c \\ {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_2 : t \\ {}^{f}P{\downarrow_4}(c, f) = t \end{array}}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_1.f = {}^{f}e_2 : t}\ \text{T-}\textsc{Write}$$

$$\frac{\begin{array}{c} {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_0 : \_ \\ {}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e : t \end{array}}{{}^{f}P, \Gamma, {}^{f}h \ \vdash_{\overline{f}} \ {}^{f}e_0; {}^{f}e : t}\ \text{T-}\textsc{Seq}$$

Figure 10: ^{f}SCHOOL type system.

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.

Theorem 4 (^{f}SCHOOL **Progress)**

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

```
class CountdownLatch {
 CountdownLatch init_CountdownLatch() {
  permits(0); return this;
 }
 int get_permits() & async permits(int r) {
  permits(r); return r;
 }
 int set_permits(int s) & async permits(int r) {
  permits(s); return s;
 }
 void await() & async countDown() {
  set_permits(get_permits() - 1);
  if (get_permits() > 0) { await(); }
 }
}
```

Listing 4: Encoded chorded Countdown Latch with fields.

Proof 7 *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.*

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 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_Countdow`
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, where field reads/writes have been replaced by calls to `get_permits` and `set_permits`. Furthermore, in listing 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 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).

Encoding Expressions

$$\begin{array}{rcl} \mathbb{C}({}^f e) & = & {}^f e \text{ if } {}^f e \in \{voidValue, \iota, null, this, x\} \end{array}$$

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

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

Encoding Programs

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

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

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

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

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 11: Encoding ^{f}SCHOOL into SCHOOL.

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

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

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 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.

$$\frac{\begin{array}{c}\forall\ \iota \in dom({}^fh)\ :\ {}^fh(\iota){=}(c,\_)\ \Longrightarrow\ h(\iota){=}c \\ \overline{e} = \{\iota.f(v) : {}^fh(\iota)\downarrow_2(f) = v\}\end{array}}{{}^fe_1, \ldots, {}^fe_n, {}^fh\ \simeq\ \mathbb{C}({}^fe_1), \ldots, \mathbb{C}({}^fe_n), \overline{e}, h}\ \textsc{C-Strong}$$

$$\frac{\overline{{}^fe}\ \cong\ \overline{{}^fe'} \qquad \overline{{}^fe'}, {}^fh\ \simeq\ \overline{e'}, h \qquad \overline{voidValue}\ \overline{e'}\ \cong\ \overline{e}}{\overline{{}^fe}, {}^fh\ \simeq\ \overline{e}, h}\ \textsc{C-Strong-Perm}$$

Figure 12: Strong Correspondence

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 preserved 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 $\overline{{}^fe}, {}^fh\ \simeq\ \overline{e}, h$, 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 $\overline{{}^fe}, {}^fh\ \approx\ \overline{e}, h$, which allows for these temporary discrepancies during evaluation of intermediate steps.

5.2 Strong Correspondence

In figure 12 we define strong correspondence between ^{f}SCHOOL and SCHOOL configurations, in the form of the judgement $\overline{{}^fe}, {}^fh \;\simeq\; \overline{e}, h$.

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 ${}^fe_i$ in the ^{f}SCHOOL configuration, its encoding $\mathbb{C}({}^fe_i)$ appears as an expression in the SCHOOL configuration.
2. All objects in the ^{f}SCHOOL heap ${}^fh$ 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 ${}^fh$.

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 employs a countdown latch (at address ι_0) where the current value of the `permits` field is 3; the ^{f}SCHOOL heap is the following:

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

and the current configuration is:

$$\iota_0.permits = 2, {}^fh$$

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:

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

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

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

$$\begin{tikzcd}
{}^f\mathrm{SCHOOL} & & \mathrm{SCHOOL} \\
\iota_0.permits = 2, {}^fh \arrow[ddd, "\textsc{Write}"] & \simeq & \iota_0.set\_permits(2), \iota_0.permits(3), h \arrow[d, "\textsc{Join}"] \\
& & \iota_0.permits(2); 2, voidValue, h \arrow[d, "\textsc{Async}"] \\
& & voidValue; 2, voidValue, \iota_0.permits(2), h \arrow[d, "\textsc{Seq}"] \\
2, {}^fh' & \simeq & 2, voidValue, \iota_0.permits(2), h
\end{tikzcd}$$

Figure 13: Example of strong correspondence.

$$\begin{tikzcd}
\overline{{}^fe}, {}^fh \arrow[r, dash] \arrow[d] & \simeq \arrow[r, dash] & \overline{e}, h \arrow[d, dotted, "*"] \\
\overline{{}^fe'}, {}^fh' \arrow[r, dotted, dash] & \simeq \arrow[r, dotted, dash] & \overline{e'}, h'
\end{tikzcd}$$

Figure 14: Soundness of the encoding.

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):

$$\begin{aligned} &\iota_0.permits = 2, {}^fh \xrightarrow{\textsc{Write}} 2, {}^fh' \\ &\quad\text{where}\quad {}^fh' = {}^fh[\iota_0 \mapsto (CountdownLatch, \{permits \mapsto 2\})] \end{aligned}$$

The corresponding initial SCHOOL configuration reaches the corresponding final configuration as shown in figure 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* 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 8 *We first prove a lemma for a single thread by case induction on the execution* $\overline{{}^fe}, {}^fh \ \longrightarrow \ \overline{{}^fe'}, {}^fh'$*; the theorem for many threads then follows.*

The diagram from figure 14 illustrates the above theorem; solid lines represent the premises and dotted lines the conclusions.

5.3 Weak Correspondence

In the example from figure 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 15 and 17 we define two forms of the weak correspondence judgement. The first has the form $\varphi \vdash {}^fe, {}^fh \ \approx \ \overline{e}, h$ and is between an individual ^{f}SCHOOL expression and a heap and potentially multiple SCHOOL expressions and a heap. The second has the form $\overline{{}^fe}, {}^fh \ \approx \ \overline{e}, h$ 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 ${}^fh \approx h$, 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.

5.3.1 Individual Expressions

The form of weak correspondence for individual expressions (figure 15) is:

$$\varphi \vdash {}^fe, {}^fh \ \approx \ \overline{e}, h$$

where the ^{f}SCHOOL expression ${}^fe$ 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

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

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

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

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

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

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

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

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

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

Figure 15: Weak Correspondence for Expressions

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 16 we illustrate the use of weak correspondence on the original example from figure 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

$$\begin{tikzcd}
\varphi & & {}^{f}\text{SCHOOL} & & \text{SCHOOL} \\
\varepsilon & \vdash & \iota_0.\mathit{permits} = 2, {}^{f}h \arrow[d, dash, dotted] & \approx & \iota_0.\mathit{set\_permits}(2), \iota_0.\mathit{permits}(3), h \arrow[d, "\textsc{Join}"] \\
\iota_0.\mathit{permits} & \vdash & \iota_0.\mathit{permits} = 2, {}^{f}h \arrow[d, "\textsc{Write}"] & \approx & \iota_0.\mathit{permits}(2); 2, \mathit{voidValue}, h \arrow[d, "\textsc{Async}"] \\
\varepsilon & \vdash & 2, {}^{f}h' \arrow[d, dash, dotted] & \approx & \mathit{voidValue}; 2, \mathit{voidValue}, \iota_0.\mathit{permits}(2), h \arrow[d, "\textsc{Seq}"] \\
\varepsilon & \vdash & 2, {}^{f}h' & \approx & 2, \mathit{voidValue}, \iota_0.\mathit{permits}(2), h
\end{tikzcd}$$

Figure 16: Example of weak correspondence.

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 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_...* 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.

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

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

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

Figure 17: Weak Correspondence for Configurations

5.3.2 Configurations

The form of weak correspondence for configurations (figure 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 asynchronous method invocations in $\overline{e}$; the last two are ensured by the following definition.

Definition 9 (Field Encoding)

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

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 ${}^fh$ 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 ${}^fh$, 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 $\varphi_i \vdash {}^fe_i, {}^fh \;\approx\; \overline{e_i}, h$).
- 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 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 20, see later section 5.3.4). 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.

We notice that after two steps of SCHOOL evaluation the φ of the top line is ι_0*.permits*, while that of the bottom line is ι_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 the φs would not be distinct, and hence the configurations would no longer weakly correspond.

5.3.3 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)

$(i)\quad \forall\; {}^fe, {}^fh, h \;:\; \varepsilon \vdash {}^fe, {}^fh \;\approx\; \mathbb{C}({}^fe), h$

$(ii)\quad \forall\; \overline{{}^fe}, {}^fh, h \;:\; {}^fh \approx h \implies \overline{{}^fe}, {}^fh \;\approx\; \overline{\mathbb{C}(e)}, h$

Proof 9 *(i) by induction on the structure of ${}^fe$; (ii) by application of* C-Wk *and use of part (i) and the definition of $\mathcal{F}$.*

Theorem 6 (Soundness of The Encoding with Weak Correspondence)

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

φ 's	^{f}SCHOOL		SCHOOL
	$\iota_0.permits = 4,$		$\iota_0.set\_permits(4), \iota_0.permits(5),$
	$\iota_1.permits = 3, {}^fh$	$\approx$	$\iota_1.set\_permits(3), \iota_1.permits(4), h$
	$\vdots$		$\downarrow$ Join
$\iota_0.permits$	$\iota_0.permits = 4,$		$\iota_0.permits(4); 4, voidValue,$
	$\iota_1.permits = 3, {}^fh$	$\approx$	$\iota_1.set\_permits(3), \iota_1.permits(4), h$
	$\vdots$		$\downarrow$ Join
$\iota_0.permits,$	$\iota_0.permits = 4,$		$\iota_0.permits(4); 4, voidValue,$
$\iota_1.permits$	$\iota_1.permits = 3, {}^fh$	$\approx$	$\iota_1.permits(3); 3, voidValue, h$
	$\downarrow$ Write		$\downarrow$ Async
	$4,$		$voidValue; 4, voidValue, \iota_0.permits(4)$
$\iota_1.permits$	$\iota_1.permits = 3, {}^fh'$	$\approx$	$\iota_1.permits(3); 3, voidValue, h$
	$\vdots$		$\downarrow$ Seq
	$4,$		$4, voidValue, \iota_0.permits(4)$
$\iota_1.permits$	$\iota_1.permits = 3, {}^fh'$	$\approx$	$\iota_1.permits(3); 3, voidValue, h$
	$\downarrow$ Write		$\downarrow$ Async
	$4,$		$4, voidValue, \iota_0.permits(4)$
	$3, {}^fh''$	$\approx$	$voidValue; 3, voidValue, \iota_1.permits(3), h$
	$\vdots$		$\downarrow$ Seq
	$4,$		$4, voidValue, \iota_0.permits(4)$
	$3, {}^fh''$	$\approx$	$3, voidValue, \iota_1.permits(3), h$

Figure 18: Example of weak correspondence with multiple fields.

$$\begin{tikzcd}
\overline{{}^fe}, {}^fh \arrow[r, dash] \arrow[d] & \approx \arrow[r, dash] & \overline{e}, h \arrow[d, dotted, "*"] \\
\overline{{}^fe'}, {}^fh' \arrow[r, dotted, dash] & \approx \arrow[r, dotted, dash] & \overline{e'}, h'
\end{tikzcd}$$

Figure 19: Soundness of the encoding with weak correspondence.

$$\begin{tikzcd}
{}^{f}\text{SCHOOL} & & \text{SCHOOL} \\
\iota_0.permits = 3, \iota_0.permits = 2, {}^{f}h \arrow[d, "\textsc{Write}"] & \simeq & \iota_0.set\_permits(3), \iota_0.set\_permits(2), \iota_0.permits(5), h \arrow[d, dashed, "?"] \\
3, \iota_0.permits = 2, {}^{f}h' \arrow[d, "\textsc{Write}"] & & \iota_0.permits(3); 3, \iota_0.permits(2); 2, h \arrow[d, "*"] \\
3, 2, {}^{f}h'' & \not\simeq & 3, voidValue, \iota_0.permits(3), 2, voidValue, \iota_0.permits(2), h
\end{tikzcd}$$

Figure 20: Example where strong correspondence is not reached.

Proof 10 *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.*

The diagram from figure 19 illustrates the above theorem; solid lines represent the premises of the theorem and dotted lines represent the conclusions.

5.3.4 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 16 and 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 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 $\iota_0.\mathit{permits}(\_)$ 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 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)

$$\varphi, {}^fe, {}^fh \sim \varphi', {}^fe', {}^fh' \text{ iff } \left\{ \begin{array}{l} {}^fe = {}^fe' \wedge {}^fh = {}^fh' \\ \qquad\quad \vee \\ {}^fe, {}^fh \;\longrightarrow\; {}^fe', {}^fh' \quad \wedge \quad \varphi' = \varepsilon \end{array} \right.$$

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)

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

Proof 11 *By structural induction on the judgement* $\varphi \vdash {}^fe, {}^fh \;\approx\; e, \overline{e}, h$ *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)

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

Proof 12 *By case analysis on the weak correspondence judgement applied to obtain $\overline{{}^fe}, {}^fh \ \approx\ \overline{e}, h$, and use of lemma 5 and definition 10. The application of the lemma yields the relationship $\varphi, {}^fe, {}^fh \sim \varphi', {}^fe', {}^fh'$ 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*).*

The left diagram from figure 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

$$\begin{tikzcd}
\overline{{}^fe}, {}^fh \arrow[r, dash, "\approx"] \arrow[d, dotted] \arrow[dr, dash, dashed, "\approx"] & \overline{e}, h \arrow[d] \\
\overline{{}^fe'}, {}^fh' \arrow[r, dash, dotted, "\approx"] & \overline{e'}, h'
\end{tikzcd}
\qquad
\begin{tikzcd}
\overline{{}^fe}, {}^fh \arrow[r, dash, "\simeq"] \arrow[dd, dotted, "*"] & \overline{e}, h \arrow[d, "*"] \\
& \overline{e'}, h' \arrow[d, dotted, "*"] \\
\overline{{}^fe'}, {}^fh' \arrow[r, dash, dotted, "\simeq"] & \overline{e''}, h''
\end{tikzcd}$$

Figure 21: Weak completeness and strong completeness of the encoding.

we can prove the following "diamond" property: if the SCHOOL expressions $\overline{e}$ are the encoding of the ^{f}SCHOOL expressions $\overline{{}^fe}$, 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 $\overline{{}^fe'}$ 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 ${}^fe$ and its direct encoding $\mathbb{C}({}^fe)$ (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)
$\varepsilon \vdash {}^fe, {}^fh \;\approx\; e, h \implies e =_{voidValue} \mathbb{C}({}^fe)$

Proof 13 *By case analysis on the derivation of* $\varepsilon \vdash {}^fe, {}^fh \;\approx\; e, h$.

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

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)

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

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

Proof 15 *By induction on the number of expressions in* $\overline{{}^f\!e}$ *and repeated application of lemma 7.*

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)

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

Proof 16 *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* $\overline{{}^{f}e'}, {}^{f}h' \simeq \overline{e''}, h''$*.*

The right diagram from figure 21 illustrates the above theorem; solid lines represent the premises and dotted lines represent the conclusions.

In this section we have shown an encoding of programs with fields using only chords. This encoding is sound, since all original program behaviours are preserved, and complete, since no new behaviours are introduced. We have shown correspondence between original and encoded program executions; however, due to concurrency and non-deterministic interleaving of expressions, we allowed a "temporary breaking" of correspondence. Such breaks are permissible, as we have shown that encoded programs reach corresponding configurations again.

6 Related work

Chords first appeared in the experimental language Polyphonic C$^{\sharp}$ [4], [5] and then in C$_{\omega}$ [3]. These are both extensions to C$^{\sharp}$, so also contain explicit thread operations and the more common synchronisation constructs (e.g. monitors). The interaction between the two concurrency paradigms would benefit from further study. There are also various library-based chorded extensions of languages such as Java [21, 20] and Scala [9].

The ideas for chords originate in the Join-Calculus [14, 15], which was implemented in the JoCaml language [12]. The Join-Calculus itself was heavily influenced by ideas from the Chemical Abstract Machine (CHAM) [6], resulting in the Reflexive Chemical Abstract Machine (RCHAM) [13].

The Join Calculus was designed to be directly implementable as a functional ML-style language, while SCHOOL is a generalised model of imperative, object-oriented languages which feature chords. There are several similarities between the Join-Calculus and SCHOOL, and some fundamental differences.

Overall, both models describe the concurrent and asynchronous execution of processes. Both the Join-Calculus and SCHOOL posit the concept of join patterns as the sole means of synchronisation between processes, and all other constructs and data structures are built on top of the joining mechanism. Join patterns in the Join-Calculus correspond with chord headers in SCHOOL.

An important difference between the two is that asynchronous concurrency is the primitive unit of execution in the Join-Calculus, and synchronous execution is achieved via continuation-passing, while in SCHOOL, execution is synchronous (with evaluation contexts), and asynchrony is obtained through asynchronous chords which become strung and spawn new processes.

A straightforward parallel is that between messages in the Join-Calculus, and invocations of methods in SCHOOL. Messages in the Join-Calculus are sent asynchronously and travel to a known destination, a channel name, which is statically bound in its immediately enclosing lexical scope. In SCHOOL, asynchronous method invocations are represented as expressions executing in parallel with the rest of the processes. The invocations are sent to known objects via the objects' addresses, and method names are statically bound to their class.

In the Join-Calculus, messages are consumed upon arrival, and paired via pattern-matching to join definitions (selection of a join pattern when more than one is possible is non-deterministic). In SCHOOL, invocations are consumed by the joining of chords, whose selection is based on pattern-matching (when more than one chord can be chosen the result is obtained non-deterministically). Hence both models are characterised by locality in terms of pattern-matching.

One significant effect of the synchronous execution of SCHOOL is that a chord can have the same method of return type void participating as both the synchronous part *and* in the asynchronous part of a chord. Consequently, pattern-matching in SCHOOL is not linear, while the Join-Calculus imposes strict linearity in join patterns.

In terms of a memory model, the Join-Calculus features a global name-space, while in SCHOOL there is a common heap between all processes. As new names can be created in the Join-Calculus, so new heap entries can be created in SCHOOL.

A fundamental difference between the Join-Calculus and SCHOOL is that the former is high-level and based on name-passing (indeed, only names can be passed), while SCHOOL has no high-level constructs and only values can be passed. As a consequence of having only names, the Join-Calculus requires all constructs to be encoded via such names, while SCHOOL features objects as first-class constructs (and ^{f}SCHOOL SCHOOL features fields as well), and consequently SCHOOL has an inheritance mechanism. Work on the Objective Join-Calculus has resulted in encodings of objects and an inheritance mechanism borrowed from ML (which uses pattern-matching on program definitions and *self*). In both models encodings of constructs necessitate naming restrictions (such as private names in the Join-Calculus and method names such as *init* in SCHOOL).

Probably the closest approach to concurrent language design, outside of the process calculus community, can be seen in Concurrent ML (CML) [18], which allows programmers to define new synchronisation and communication abstractions as first-class, thus giving flexibility to tailor concurrency abstractions.

There are many different ways that concurrency is expressed in object-oriented languages. In Simula [10], for example, objects were processes and messages were passed between them. There are several excellent surveys describing the different paradigms [8, 17].

Probably the oldest model similar to chords are the active objects [7] of actor languages [2], that combine objects and processes. Actors use asynchronous message passing and continuations to provide concurrency, and state is held in the messages, as with chords. However, a message sent to an Actor has an explicit method destination, while with chords, it is the *join* semantics and scheduling which ultimately decide which chord consumes a message.

7 Conclusions and Future Work

The model of chords presented in this paper aims to provide a simple and formal basis for investigating the crucial aspects of chorded language design, with a focus on the semantics of *join* in object-oriented languages.

Our model enables us to reason about the interaction of other language constructs with chords (as we have shown with fields in this paper), and argue about the relative expressive power, and hence utility, of including constructs in a chorded language.

SCHOOL does not attempt to model specific chorded languages, rather, it is intended to be an abstract and flexible model of chorded languages in general. There are valid SCHOOL programs which are not valid in Polyphonic $C^\sharp$, since SCHOOL abstracts from limitations imposed on Polyphonic $C^\sharp$ programs for avoiding inheritance issues, restrictions not necessary for proving subject reduction and completeness.

In this paper we have investigated the interaction of chords with fields. We have found that chords suffice to fully encode the behaviour of programs using fields, and hence fields to not add expressive power to chorded languages. In order to demonstrate this we have shown correspondence between ^{f}SCHOOL, an extension of SCHOOL with fields, and an encoding of fields using only chords. Our encoding preserves the behaviours of original programs and does not introduce new behaviours.

As chords are a construct within a conventional object-oriented language which already contains more traditional concurrency constructs, such as monitors and explicit threads, we would like to obtain similar results for these. Our more long-term ambitions are to investigate and formalise the properties of larger, cross-cutting issues of chorded programs such as exception handling, scheduling, and analysis and verification (such as detection of progress, lack of deadlock or livelock, and so on).

Acknowledgements

We would like to thank John Knottenbelt for his solution to the Countdown Latch, Alex Buckley for his work on the original model of SCHOOL_0, Feruccio Damiani for his suggestions on the formalism, Paola Giannini for her insight which lead to the unified treatment of synchronous and asynchronous chords (and a much simpler semantics without explicit queues), and Neil Datta and Anastasia Niarchou for their general comments.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag New York Inc., 1998.

[2] Gul Agha and Carl Hewitt. Concurrent programming using actors. In *Object-oriented concurrent programming*, pages 37–53. MIT Press, 1987.

[3] Nick Benton, Gavin Bierman, Luca Cardelli, Erik Meijer, Claudio Russo, and Wolfram Schulte. C_ω. `http : //research.microsoft.com/Comega/`, 2004.

[4] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for $C^\sharp$. In B. Magnusson, editor, *Proc. of ECOOP02*, volume 2374, pages 415–440. Springer Press, 2002.

[5] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for $C^\sharp$. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[6] 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.

[7] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86. ACM Press, 1986.

[8] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[9] Vincent Cremet. Join definitions in scala. `http : //lamp.epfl.ch/ cremet/join_in_scala/`.

[10] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[11] 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.

[12] 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.

[13] 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.

[14] Cédric Fournet and Georges Gonthier. The join calculus: a language for distributed mobile programming. *Applied Semantics Summer School*, pages 1–66, 2008.

[15] 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.

[16] Demetrios Nicolaou. A virtual machine for a chorded programming language. Master's thesis, Department of Computing, Imperia College London, 2007.

[17] Michael Philippsen. A survey on concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, August 2000.

[18] 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.

[19] Lefteris Volanakis. Implementation of a chorded compiler. Master's thesis, Department of Computing, Imperial College London, 2006.

[20] 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.

[21] Tim Wood. A chorded compiler for java. Master's thesis, Imperial College, London, June 2004.