**Imperial College of Science,
Technology and Medicine
(University of London)
Department of computing**

Comparing the Expressive Power of Monitors and Chords

by

Dave Cunningham

**Submitted in partial fulfilment
of the requirements for the MSc
Degree in Advanced Computing of the
University of London and for the
Diploma of Imperial College of
Science, Technology and Medicine.**

September 2005

Abstract

The report presents a formal argument for the relative expressive power of monitors and chords, when used for synchronisation in concurrent, imperative, object-oriented languages. Our formalism of expressive power is based on the work of Matthias Felleisen[5]. We review and then formalise monitors and chords. We use these formalisations to formally prove properties as in [5].

First we show that asynchronous chords can be removed from a chorded object oriented language and replaced with a `spawn` construct without affecting the expressiveness of the language. This means the programmer can use `spawn` instead of asynchronous chords, for thread creation, without affecting the structure of the program. We use this result to compare languages with just synchronous chords, to Java-like languages with monitors.

We show that chords and monitors have equal expressive power. We conclude from this that the introduction of chords into real programming languages are unlikely to revolutionise programming practice.

Keywords:

- Programming language
- Object oriented
- Concurrency, Concurrent, Multithreaded
- Synchronisation
- Expressiveness, Expressibility
- Chords, Message passing
- Lock, Monitor, Mutex
- Operational semantics

Acknowledgements

I must sincerely thank my supervisor Dr Sophia Drossopoulou for her enthusiasm and guidance during this course. Without the hours she spent advising me in person, as well as by email, and without her detailed comments on all the writing that she encouraged me to do, I fear this project would not have been possible.

I am also very grateful for the support Dr Nobuko Yoshida gave, during a few weeks while Sophia was away. Her enthusiasm and expertise on concurrency theory was much appreciated.

Contents

Chapter 1

Introduction

1.1 Motivation

It is widely believed that in the near future, many more concurrent programs will need to be written[11]. As with any other programs, they need to be written well, and it is an accepted fact in the field of Software Engineering that good program design can encourage readability (allowing easier maintenance) and discourage bugs.

We know that some languages allow program designs that other languages do not, we say a language is *more expressive* than another. This was elegantly formalised in [5]. We attempt to adapt this technique to reason about the expressiveness of two approaches towards concurrent programming language design. If a language cannot express a particularly good class of program designs, then that language would not be a good choice for programmers.

Concurrent programs have a fundamentally different model of execution to sequential programs, and in particular there is a requirement for different threads of execution to affect each others' progress, i.e. they occasionally need to synchronise. In this report we compare two fundamentally different approaches towards providing this utility to the programmer:

- We can synchronise two threads with a system of monitors. This approach is older than chords, and was originally documented in [8]. It has been conventionally employed by operating systems and real concurrent languages to date, e.g. POSIX[9] and Java[7]. Languages using monitors typically have two complementary features: The mutual exclusion primitives are intended for preventing any thread from executing some code if some other thread is executing that code. Condition variables allow one thread to suspend its execution until another thread lets it continue.

- We can synchronise two threads with a notion of message passing called *chords*. Its obvious application is for defining communication between threads, i.e. the transfer of run-time data from one thread to another. Additionally, because the receiving thread cannot proceed without this information, it is effectively being stalled, and thus the execution of one thread is affecting the progress of another. There has been recent research[2] towards the use of chords in the practical programming language $C^\sharp$.

In this report we show that these two approaches are equally expressive. There are no program designs that are only expressible with one or the other. We conclude that chords are unlikely to revolutionise concurrent software development in the same way that adding recursion and exceptions to languages has done. Neither does exclusively using chords have the same benefits as exclusively using structured control flow or garbage collection.

Chords may, however, allow certain complex concurrent programs to be written more concisely, and thus their impact may be similar to the impact of object oriented features into a language (not to be confused with using a language like C to write an object-oriented program using function pointers to implement late binding and so on). At this stage, is hard to say of what magnitude the impact of chords would be, but we hope to have determined what *kind* of impact it would be.

1.2 Structure of report

The remaining chapters of this report are as follows: Chapter 2 summarises the context in which this work exists. This includes the external ideas and adaptions thereof that were used in this report (such as the notion of expressiveness from [5]). We also describe in detail the synchronisation constructs that this report compares, i.e. monitors and chords. Chapter 3 explains the way in which we tackle the problem of measuring expressiveness in this report, with reference to the content of the previous chapter.

The main body of the report, chapters 4, 5, 6, 7, and 8 contain the definitions, theories, and proofs that substantiate the conclusion. Although this work compares only two different approaches to synchronisation, there are many intermediate languages and a path is defined through these languages to connect the two paradigms. Chapter 3 contains more detail on the structure of the body of the report.

Chapter 9 summarises and consolidates the results of the body of the report and builds from this a conclusion. Chapter 10 criticises the method and techniques used in the report, and discusses the value of the results. There are also some suggestions for further work.

Chapter 2

Background

2.1 Expressiveness

Much of the work in this report uses the formalisation of programming language expressiveness from [5]. In fact, [5] defines two concepts of language expressiveness, we use the stronger notion of "macro-expressiveness" throughout. The approach in [5] was to define translations between languages that preserve three properties. The first two properties were program *validity* (this was called programness) and program *behaviour* (this was simplified to termination behaviour).

A third property was needed because the languages considered were Turing-complete, there was always an equivalently behaving and valid program in the target language. Sometimes, however, the translation had to massively change the structure of the program it was translating, in order to make it "fit" into the target language, and this was considered unacceptable. To reject translations that changed the general structure of the program they were translating, it was required that the third property, "structure-preservation", was preserved over the translations. The existence of such a translation implied that source language was at most as expressive as the target language.

Actually, the method in [5] was to demonstrate that certain constructs were "eliminable". Constructs were removed from a language and it was shown (through definition of an appropriate translation) that the restricted language was at most as "expressive" as the original language. This means that no program designs were lost as a consequence of *restricting* the language.

The translation had to preserve program *validity* and *behaviour*, but more importantly, program *structure*. It was the definition of this property of structure-preservation where the particular contribution of [5] lay. A translation $\varphi : \mathcal{L}^1 \rightarrow \mathcal{L}^2$ where $\mathcal{L}^2 \subset \mathcal{L}^1$ was structure-preserving if and only if:

- It was homomorphic over program constructs that were not being eliminated, i.e. the constructs in $\mathcal{L}^2$. This includes code in which constructs that *are* to be eliminated are nested, and also code that is nested inside such constructs. We must visualise the eliminated constructs as a set of specific points in the syntax tree, and the translation must only affect those points, it must preserve the structure (or shape) of the rest of the tree.

- Its effect on eliminated constructs can be summarised as follows: If c is an eliminated construct(i.e. $c \in \mathcal{L}^1$ but $c \notin \mathcal{L}^2$) with an arity of n, and $c(e_1 \dots e_n)$ is a piece of program syntax, then $\varphi(c(e_1 \dots e_n))$ can be described as a "macro" encoding of $\varphi(e_1) \dots \varphi(e_n)$. This means that the result of encoding $c(e_1 \dots e_n)$ is a piece of syntax in which $\varphi(e_1) \dots \varphi(e_n)$ are contained, and whose structure is otherwise independent of $\varphi(e_1) \dots \varphi(e_n)$. Crucially, the $e_1 \dots e_n$ must remain intact, aside from the recursion of the translation.

For example, it can be shown that the Java's `for` construct can be eliminated from Java using a simple macro translation that uses `while`. However, the language feature of *exceptions* cannot be represented in terms of method return values, since the translation would have to change the code that invokes the method, the code that invoked *that* method, and so on, all the way back to the place where the exception is caught. This would clearly not be structure-preserving since it requires the modification of many areas of the syntax tree that are not places where the exception syntax is used. We can conclude that Java without `for` is as expressive as Java with `for`, however a Java without exceptions is less expressive.

2.2 Synchronisation via monitors

Monitors were described in [8], and despite there being other mechanisms available for expressing synchronisation (such as various forms of message passing in functional languages, and the Ada approach), they are by far the most common used in practical programming languages. We could speculate that the reason for this is their very fundamental nature means they are very versatile, and also easy (and efficient) to implement.

As mentioned in section 1.1, there are two complementary mechanisms in any monitor implementation, mutual exclusion (which ensures no two threads are executing a "critical section" of programming code) and condition variables (which allow a thread to tell another thread to progress). Different programming languages use different terminology, and the exact syntax and "presentation" to the programmer has changed, but the underlying concept is the same.

- Programmers often casually speak of "locks", when referring to a wide variety of mechanisms such as placing files on the disk to let other programs know that they are running, and internal file-system mechanisms for preventing multiple programs from accessing the same file, as well as the monitors found in programming languages.

- The POSIX standard[9] talks of mutexes, which is an abbreviation of "mutual exclusion", and condition variables. Mutual exclusion requires us to mark critical sections in the code, and in POSIX we do this by calling a specific function at the beginning and end of a critical section. Condition variables are implemented with the functions `pthread_cond_wait()`, `pthread_cond_signal()` and `pthread_cond_broadcast()`.

- Older versions of Java[7] have a `synchronized` construct that encloses the critical section, and thus is a higher-level interface to mutual exclusion. They also have the object methods `wait()`, `notify()`, and `notifyAll()`, which are an implementation

of condition variables. Newer versions (since 1.5) also have POSIX-style mutexes where critical sections can be delimited with specific function calls.

We will in future refer this to synchronisation approach (the two complementary mechanisms) as *monitors*. Mechanisms for implementing mutual exclusion will be referred to as *mutexes*. The mechanism which is called "condition variables" in [8] and POSIX[9], we will refer to by same name, but when we refer to the individual methods we will use the Java style, i.e. `wait()`, `notify()`, and `notifyAll()`, because Java has become a more widely used language in academia.

Studying monitors gives us insight into many more languages, since the concurrency features found in many other programming languages and operating system libraries are very similar to monitors. Most languages simply provide a compatibility interface to the underlying operating system's libraries, of which many use the POSIX approach as described above.

We now discuss the precise behaviour of mutual exclusion, and look at two solutions to the problem of specifying the critical section in a program. Then we look at re-entrant mutexes which are a higher-level mutex implementation that provide a more intuitive behaviour. We then consider the precise behaviour of condition variables and the role they play in synchronisation. Finally we look at how threads are spawned, since in order to have a concurrent program, we need to spawn threads.

2.2.1 Mutual Exclusion

Motivation

Mutual exclusion is the effect of preventing two threads from executing a particular block of code concurrently. This block of code is often called a critical section and will typically cause some inconsistent intermediate state in shared memory, that must not be exposed to other threads. We can define inconsistency of shared memory by saying that some invariant is broken.

Such intermediate states are a natural consequence of composing a sequence of primitive commands to define an operation on a data-structure. Before and after the sequence of operations, the invariant holds, but it must be broken *during* the operation.

Without concurrency there is no problem since one can abstract such a sequence of commands in a function or method call. The calling function does not see any intermediate state of the operation, it only sees the state before and after the function call, where the invariant holds. Because of this, operations on data-structures often assume the invariant holds at the start and end of the sequence of commands.

When a thread is part-way through the execution of an operation on a data-structure, and another thread begins an operation on the same data-structure, it is not valid to assume that the invariant holds. This causes bugs called "race conditions" that are hard to locate because their occurrence depends on the mutual state of two threads which is non-deterministic.

Using mutual exclusion, however, we can ensure that one operation has finished before another begins, thus we can ensure the invariant holds. Thus we hide the intermediate states caused by other threads interacting with the data-structure, in the same way that functions encapsulated the intermediate state in a non-concurrent environment.

Mutual exclusion is achieved by preventing the interleaving of one thread with another, while they are executing related critical sections. If *aaaa* is the sequence of four operations that implements some operation on a data-structure, and *bbbb* is the same sequence on the same data, executed by another thread, mutual exclusion permits only the interleavings *aaaabbbb* or *bbbbaaaa*. The other sixty eight interleavings are not permitted. We say that the critical section is atomic, since no thread can observe any intermediate states which would divide the critical section into two parts.

Examples of use

As a contrived example, we consider the shared memory data-structure of a pair of integers (m, n), and the operation of assigning the same value to them both. If the following code was executed concurrently, by two threads, there would be many more behaviours that we would intuitively expect, particularly, the invariant that $m = n$ would not be maintained:

```
local x := random();
m := x;
n := x;
```

If one thread binds `x` to 4, and the other thread binds `x` to 5, then the first thread executes `m := 4`, then the other thread executes `m := 5` and then `n := 5`, but then the first thread continues and executes `n := 4`, then the resultant state is $m = 5$ but $n = 4$. To prevent this, we specify that the code `m := x ; n := x` is a critical section. The approach for doing this varies from language to language, we use the POSIX approach in the following example:

```
local x := random();
lock()
m := x;
n := x;
unlock()
```

The effect of this change to the code on the concurrent execution, is to delay the execution of `m := 5` until after `n := 4`. If the second thread gets the lock first, the execution of `m := 4` will be delayed until after `n := 5` has finished executing. Either way, the possible results obey the invariant that $m = n$. The state where a thread has assigned to `m` but not to `n` is not "visible" to any other thread.

Mutexes are intended to hide the intermediate state of computations on a shared data-structure, by synchronising different threads such that they are not executing the critical sections associated with that data-structure concurrently. Typically we enclose all code whose correctness depends on the consistency of some shared data-structure, whether it be reading or writing, with the same mutex, so only one thread can interact with the data-structure at a time.

2.2.2 Specifying a critical section with `lock()` and `unlock()`

We can specify a critical section by calling a function `lock()` just before the start of the critical section, and `unlock()` immediately after the end. This is the approach used by

POSIX [9], because POSIX is an extension of the C language that is purely implemented with libraries, so an extension of the C syntax was not possible. The implementation of the functions `lock()` and `unlock()` implements the actual mutual exclusion.

When the call to `lock()` returns, we say that the thread *owns the lock* on that mutex, or *has locked* that mutex. This state persists until `unlock()` is called by the same thread.

It is common to have many different mutexes being used concurrently, the mutex itself being an opaque data-structure passed to the `lock()` and `unlock()` with an argument. Thus, one can lock, or unlock a specific mutex. This means that the set of critical sections is divided into groups that are each guarded by a mutex, and no two threads can be concurrently executing a critical section of the same group.

If we used the same mutex for all the groups (effectively squashing the groups into one big group), then no two threads could be in *any* critical section. Normally this preserves the correctness of the program, but the unnecessary stalling of threads makes the execution less efficient.

We call this effect on synchronisation the *granularity* of synchronisation, high granularity means there is too much mutual exclusion, too little parallelism, and unnecessary low efficiency. It is caused by having too few mutexes as described above, but also by marking critical sections larger than they need to be.

Here is an example of delimiting a critical section with the function calls described above:

```
lock(myMutex);
/* critical section (code that runs with the lock on myMutex) */
unlock(myMutex);
```

The behaviour of each function, when called by a thread, depends on the state of the other threads in the system. Calls to `lock()` will stall unless no other thread has the lock on the specified mutex. Calls to `unlock()` never stall, but will wake up a thread that was waiting to enter, should one exist.

The stalling effect of `lock()` occurs if any thread has the lock, including the calling thread, so for example, executing `lock();lock()` will block. The behaviour of calls to `unlock()`, when the calling thread does not own the lock, is undefined, and valid programs never do this.

In summary, calls to `lock()` attempt to obtain a specific lock. If that lock is already taken, the call blocks until it is safe to obtain it. Calls to `unlock()` release the lock, and always return immediately.

2.2.3 Specifying a critical section with `synchronized`

Initially, Java did not expose `lock()` and `unlock()` methods directly to the programmer. It was decided that it was too easy for the programmer to forget to call `unlock()`, causing deadlock, analogous to the way that if the programmer forgets to call `free()`, dynamically allocated memory will be "leaked". It was thought that it would be better if mutexes were supported with a syntactic construct like `if`, where the critical section would be specified as part of the syntax. Consider the following program:

```
lock(myMutex);
/* code that runs with the lock on myMutex */
unlock(myMutex);
```

Using Java's `synchronized` keyword, this would be:

```
synchronized (myMutex) {
        /* code that runs with the lock on myMutex */
}
```

This is called structured synchronisation, as opposed to un-structured, because the difference is analogous with the difference between using the lower level `goto` statement to define "unstructured" control flow instead of the more syntax oriented `if` and `while` constructs.

We cannot directly use structured synchronisation to implement certain algorithms, for example this solution to the readers/writers problem requires the locking and unlocking of mutexes in unrelated parts of the syntax tree:

```
void read_lock() {
        lock(counter);
        if (num_reading==0)
                lock(main);
        num_reading++;
        unlock(counter);
}

void read_unlock() {
        lock(counter);
        num_reading--;
        if (num_reading==0)
                unlock(main);
        unlock(counter);
}

void write_lock() {
        lock(main);
}

void write_unlock() {
        unlock(main);
}
```

This solution is a higher-level mutex. Client code would use the "write lock" to delimit critical sections that write to a shared data-structure, but use the "read lock" to delimit critical sections that only need to read the state. This decreases the granularity because multiple threads can read the shared data-structure concurrently without losing correctness.

We need mutual exclusion to implement this mutex, since the higher-level mutex itself is a shared state, operations on which are a sequence of primitive commands. We cannot

use `synchronized` however, because that requires critical sections to be enclosed neatly by the syntax, whereas in our above implementation the critical section begins in one function call, continues in the calling function, and then is ended in a completely different function.

We can however implement `lock()` and `unlock()` functions, in the style of the POSIX mutexes, using condition variables to implement the stalling and restarting of threads that is required. This means that structured synchronisation does not restrict the expressiveness of Java, but it may be inconvenient to have to implement the following code when it is required.

```
class Mutex {
        boolean engaged = false;
        synchronized void lock() {
                if (engaged) wait();
                engaged = true;
        }
        synchronized void unlock() {
                engaged = false;
                notify();
        }
}
```

Moreover, a mutex implemented in this fashion is disjoint from the mutex that is associated with each object. We can either use `synchronized` for all critical sections, or use `synchronized` to create a new mutex, and use the new mutex for all critical sections. The syntax of `synchronized`, particularly the syntax sugaring that allows it to be embedded in a method type signature, allows for some very elegant code so it is not a good thing to be forced to use the above method for all the synchronisation in a class.

It was decided for later versions of Java to include unstructured synchronisation as well as the `synchronized` keyword[3]. This was implemented with library functions in the style of `lock` and `unlock`. It is still recommended for programmers to use `synchronized` where they can, and only use the unstructured synchronisation features when they absolutely have to. The library mutexes are completely disjoint from the mutexes embedded in objects, so `synchronized` cannot be used with them.

2.2.4 Re-entrant mutexes

Re-entrant mutexes are a higher-level mutex, they can be implemented on top of non-re-entrant mutexes. The interface is the same (we delimit critical sections as usual) but their behaviour is more intuitive. The catch is that the logic that defines this behaviour is more complex.

All Java's concurrency primitives and libraries use re-entrant mutexes[7, 3]. POSIX requires that implementations support both types of mutexes. Re-entrant mutexes are called *recursive* mutexes in POSIX, the other kind of mutexes (described above) are called *fast*.

According to its manual page[10], the default behaviour for the "Linuxthreads"[1] mutex

[1]Linuxthreads is an implementation of the concurrency features of the POSIX standard.

implementation is *fast* as opposed to *recursive*. POSIX requires that both are supported, but the default is undefined: [9]

> "If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behaviour."

The difference between non-re-entrant and re-entrant mutexes is in the behaviour of calls to `lock(myMutex)` by threads that already have the lock on `myMutex`. At run-time this can be seen as "re-entering" the critical section, and in the source of a program, as defining critical sections that are nested inside each other.

With a non-re-entrant mutex, the second call to `lock()` blocks. Since the only thing that will *wake* it is a call to `unlock()`, by the thread that owns the thread (i.e. the *blocked* thread), the thread will be remain blocked forever. Even worse than this, any other threads that were waiting to get the lock will also be stalled forever. In fact, this is an example of deadlock, and deadlock is never a useful behaviour.

With a re-entrant mutex, when a thread with the lock acquires the lock again, a counter is incremented, and the call returns immediately. The thread has to unlock the mutex the same number of times that it locked it, in order for it to be actually released so that other threads can proceed. This is intuitive because we want the mutex to implement mutual exclusion, that is we want there to be a maximum of one thread in the critical section, so we only need exclude *other* threads, we do not need to exclude ourself!

This is useful because we may not know whether or not we have the lock already. In fact designing the program so that ownership or not of the available mutexes is always evident at each point of the program, is very difficult. In the following example, we may like to call `g()` without already having the lock, so we let the body of `g()` acquire it, but this is impossible without re-entrant mutexes because then calls of `g()` by threads that actually do have the lock will deadlock.

```
void f() {
        lock(myMutex);
        critical section...;
        g();
        critical section...;
        unlock(myMutex);
}
void g() {
        lock(myMutex);
        critical section...;
        unlock(myMutex);
}
```

A common idiom is for functions like `g()` to *not* acquire the lock, even though they need it, and for the function documentation to mention that the caller must have the lock before calling that function. This is exposing the synchronisation (that is inherent to the algorithm) to the caller, breaking encapsulation. Java's documentation for `wait()`, from the API reference, requires us to have the lock before calling the function: "The current thread must own this object's monitor". This is strange, however, since Java has re-entrant

locks, and could instead have just acquired the lock from inside `wait()`. Instead I have often used the following pattern in programs, which wastes space:

```
synchronized(o) { o.wait(); }
```

If re-entrant mutexes are more intuitive, and promote more encapsulation of code, why do non-re-entrant mutexes persist? Because if the mutex does not have to maintain a counter, the implementation of that mutex can be much more efficient.

Perhaps we should use static analysis to detect when a mutex is not "recursively" locked, and use the faster mutex in this case[2], but this is not the subject of this report. Sometimes, there are run-time checks to see if a non-re-entrant mutex is locked twice by a thread, which can be enabled while the programmer is debugging.

It is possible to implement re-entrant locks on top of non-re-entrant locks in the following manner, so they do not add to the expressiveness of a language by the definition of [5]: (this is also the way that Linuxthreads implements recursive mutexes)

```
void r_lock(mutex) {
        if (mutex.thread==currently_running_thread) {
                mutex.counter++;
        } else {
                lock(mutex.real_mutex);
                mutex.thread := currently_running_thread;
        }
}
void r_unlock(mutex) {
        mutex.counter--;
        if (mutex.counter==0) {
                mutex.thread := null;
                unlock(mutex.real_mutex);
        }
}
```

Note that in the above implementation, `currently_running_thread` allows us identify and compare the various threads at run-time, in the algorithm itself. Therefore we need the language to provide this feature in order for this algorithm to be expressible.

2.2.5 The behaviour of condition variables

In both Java, and POSIX, there are three functions that implement a kind of low-level synchronous message passing. The functions are `wait()`, `notify()`, and `notifyAll()`. Calling it a kind of synchronous message passing is a bit of a stretch because no information is actually transferred. But aside from this detail, we can imagine that a thread calls `wait()` to receive a message, which blocks until another thread calls `notify()` or `notifyAll()`.

[2]This would not locate all the places where a faster non-re-entrant mutex can be used without deadlock, but it would still be better than using the slower re-entrant kind for everything.

The basic idea is that one thread writes the information into shared memory, while the receiving thread waits. When the information has been written, the writing thread notifies the receiving thread, which then reads the information out of shared memory, and both threads then continue as normal. The difference between `notify()` and `notifyAll()` is that if there is more than one thread waiting, `notifyAll()` will restart all of them, whereas `notify()` picks an arbitrary thread to restart.

In the example below, the "message" consists of a pair of integers, stored in the shared variables `a` and `b`:

```
thread1() {
        ...
        wait();
        printf("a+b="+(a+b));
}

thread2() {
        ...
        x := random();
        a := x;
        b := x;
        notify();
        ...
}
```

Actually, there is a relationship between the mutexes and the condition variables. One waits upon and notifies a specific mutex, which is specified in the argument of the calls. In Java, every object has a mutex embedded within it, so one calls `obj.wait()`. In order to call any of the three condition variable functions, you must possess the lock on the associated mutex[3]. In the example above, this is not inconvenient because if more than one thread is writing a message to the shared memory, there would be a race condition. Likewise, if one thread is reading the message, the writer needs to wait until the reader has finished reading, so we have to protect the variables a and b anyway. We refine the example:

```
thread1() {
        lock(mutex);
        ...
        wait(mutex);
        printf("a+b="+(a+b));
        unlock(mutex);;
}

thread2() {
        ...
        x := random();
```

[3]This is actually an implementation detail which has been unfortunately (and unnecessarily) exposed to the programmer. Using re-entrant mutexes we can implement these functions so that they can take the lock themselves.

```
        lock(mutex);
        a := x;
        b := x;
        notify(mutex);
        unlock(mutex);
        ...
}
```

To make this work, we need the call to `wait()` to give up the lock as it enters the waiting state. Otherwise no thread could actually be notified, since the thread doing the notifying would need the lock, but the waiting thread would still possess the lock. However, the waiting thread has to lock the mutex again when it wakes, before it can continue. This effectively divides the critical section into two atomic parts, so we must make sure any assumed invariants hold when `wait()` is called. When `notify()` is called, the lock is not given up, so the threads that are awoken do not actually make any progress until the notifying thread gives up the lock at the end of its critical section.

In summary, condition variables allow one thread to synchronously communicate with another thread in a very low level, fundamental way. As we shall see, this behaviour is sufficient to encode the more high-level chords, as well as many other kinds of message passing. We conclude with the following remark from [8], which hints at the potential for less fundamental constructs such as chords.

> "This design of the condition variable has been deliberately kept as primitive and rudimentary as possible, so that it may be implemented efficiently and used flexibly to achieve a wide variety of effects. There is a great temptation to introduce a more complex synchronization primitive, which may be easier to use use many purposes. We shall resist this temptation for a while."

2.3 Synchronisation via chords

Chords are a recent idea, proposed in an extension of C♯ called Polyphonic C♯[2]. The semantics of chords is inspired by the join calculus[6]. Chords synchronise through message passing, and thus there is a dualism between synchronisation and data. The basic idea is that instead of one thread invoking a method body by calling the method (as is conventional), several threads invoke a chord body, by each calling a separate method. A chord is therefore a set of methods and a chord body, instead of a single method and a method body.

```
int f(int a) & async g(int b) & async h(int c) {
        return a+b+c;
}
```

The arguments from all of the method calls come together to instantiate the chord body, which is executed either in a new thread (for asynchronous chords) or in the thread that called the synchronous method (for synchronous chords) in a similar manner to conventional methods. We can consider the method calls to be passing their arguments as a message, and since this happens between threads, this is therefore an example of inter-thread communication.

2.3.1 Behaviour

A chord is either an asynchronous chord, in which case we say all its methods are asynchronous, or it is a synchronous chord, in which case one of its methods is synchronous, and the rest are asynchronous. The above chord is synchronous, the following chord is asynchronous.

```
async g(int b) & async h(int c) {
        printf(b+c);
}
```

When an asynchronous method is called, the argument is held in a temporary, unordered queue, and control immediately returns. The queue stores the asynchronous method's argument values until they are used in a chord body. Thus, all asynchronous methods have an associated queue. In an asynchronous *chord*, where all the methods are asynchronous, the chord body will be invoked in a new thread when there is a message available in the queue for each of its constituent methods. The chord is said to be "strung". The choice of which element popped from a queue when this occurs is non-deterministic, i.e. the queue is not a "fifo".

When a synchronous method is called, the thread may stall. Synchronous methods exist as part of any number of synchronous chords. Let us take any synchronous chord of which it is a member (and thus all the other members of that chord are asynchronous). The behaviour of the call is dependent on the state of the queues of all the asynchronous methods in that chord.

- If there is a message available on each queue, the body of the chord will be invoked in the thread that called the synchronous method, just like a conventional method invocation, and an argument will be popped from each queue for the body to use.
- If there are not sufficient messages available to instantiate the chord body, the method call blocks, and the calling thread enters a "wait queue". At some later time, when at least one message has been pushed onto each of the required queues, the thread is woken up and the chord body is invoked as above.

The call only blocks if none of the synchronous chords, of which the synchronous method is a member, have a full complement of non-empty queues. If there is more than one contending synchronous chord whose asynchronous method queues are non-empty, an arbitrary chord body is invoked (another example of non-determinism in chord semantics).

It is the synchronous chords that provide actual synchronisation, the asynchronous chords are simply a convenient way to start new threads. One key aspect of chords is that the act of calling methods is tightly coupled with synchronisation. A more detailed introduction to chords (with much advocacy) can be found in [2], and also in [4], together with many examples.

2.3.2 Example

Here, we present an example of an application which is ideally suited to a chorded implementation. This is slightly adapted from [12, 2]. The reason this application is appropriate

for chords, is that its requirements for the behaviour of asynchronous message passing happen to be identical to the semantics of chords. The queueing of messages is correct, and message passing is coupled neatly with method invocation. Thus, the code need only act as a thin intermediate layer between the semantics and the user.

An "active object" is an object that has associated with it its own thread. Conventionally, when an object's methods are called, the method body is invoked into the thread that called the method. When an active object's methods are invoked, the effect is asynchronous – the arguments are passed to the object's thread and the body is executed there. Execution of the body is deferred. This means that only one thread can be executing any of the object's methods at any one time. This can be implemented in chords as follows: First we have an abstract base class that implements no messages:

```
public abstract class ActiveObject extends Thread {
        abstract void processMessage();
        void run () {
                while (!done) { this.processMessage(); }
        }
}
```

We now extend this object and pair the processMessage method with an asynchronous method for each message the active object is to receive. The effect of this is that other threads can call e.g. `addClient(c)`, which will immediately return, this will then join with some blocked or future `processMessage()` call from the object's thread, where the body of the `addClient` chord will then be executed.

```
class StockServer extends ActiveObject {

        void processMessage() & async addClient(Client c) {
                /* process addClient message */
        }

        void processMessage() & async wireQuote(Quote q) {
                /* process wireQuote message */
        }

        void processMessage() & async closeDown() {
                /* process closeDown message */
        }

}
```

2.4 Spawning threads

In POSIX, threads are spawned with `pthread_create`. A pointer to a function is supplied as an argument to this function, and the run-time environment then executes the referenced function in a new thread. In Java, threads are wrapped in a `Thread` object and the actual creation of a new thread occurs when `myThread.start()` is invoked, the method

`myThread.run()` is executed in the new thread (classes inheriting `Thread` must define this method). If we consider a `Thread` object with an overridden virtual method `run()`, this is equivalent to having a pointer to a function, so these techniques are very similar.

We do not model the `Thread` object since it is not used for synchronisation. It is really an artifact of Java's "everything is an object" aesthetic. We also do not model pointers, although we deal with references to objects on the heap which are in some ways equivalent. We could have modelled the invocation of a new thread with a construct like `fork myObject`, and require that myObject implement a method called `run`, but there is a simpler solution. We can use a construct that is not present in many realistic languages, but is equivalent to constructs that are.

In order to model the asynchronous invocation of code in a new process, we use a `spawn` e construct in the language, which explicitly specifies the actual code that will execute in the new thread, rather than implicitly using some convention of a `run()` method to hold this code. We claim that `spawn` e is equivalent to:

```
(new Runnable() { public void run() { e; } }).start()
```

This uses an anonymous inner class, which is something we do not model. There are also problems with using locally defined variables (that are not of final type) within `e`, which we also do not model. However, we believe that this is an adequate model of the thread-creation features of real languages for our purposes.

Chapter 3

Approach

We want to draw conclusions about the relative expressiveness of chords and monitors, or more specifically, of languages that use chords and monitors. Using the technique from [5] on a realistic programming language would be possible, but a great deal of work. The majority of such work would be uninteresting as it would not relate to the synchronisation issues in programming.

Instead we prove properties of abstract programming languages that are much simpler. We believe that the results scale up to realistic languages, and this will be discussed below. An additional benefit is that our results will apply to a large number of realistic languages, since many use the same fundamental approaches to concurrency despite being otherwise very different.

The abstractions chosen are object-oriented languages. We must be sure that the nature of inheritance does not interact badly with the kinds of changes that need to be made to programs when porting them to a different kind of concurrency platform. Concurrency has been called the next programming revolution after object-oriented programming, so it makes sense to build from this platform.

Our simplified models of real programming languages can be thought of as missing some of the less relevant language features, e.g. exceptions, local variables, etc. Our expressiveness results scale automatically to languages including *some* of these features, but others require extensions to our proofs.

The results automatically scale to more realistic languages that are equally expressive because of the transitivity of the expressiveness relation (it is an equivalence relation). For instance if we add input/output to the formalisation, the expressiveness is not affected since we could always have simulated input/output with method calls. Thus, we know that our results will extend to languages with input/output.

Other features, such as exceptions, would have to be subjected to further proof in the context of our languages because they increase the expressive power of the language. Only then can we can be sure that the same results apply. We believe that doing so would not present any significant problem as the only reason adding features would break the existing proofs, is if they somehow interfered with synchronisation, and we have modelled all the features that we believe would be capable of this.

We extend the object-oriented foundation with the different approaches for synchronisation (chords and monitors) to create different languages. We then define translations between these languages that preserve structure, validity and behaviour. The existence of

each translation is proof that the source language was at most as expressive as the target language, and we collect these results together to form a conclusion about the relative expressiveness of chords and monitors.

Figure 3.1 shows an overview of all the languages and translations defined in this report. Directed arrows denote formally defined translations, waving lines denote an assumed (but not proven) equivalence, i.e. the languages concerned are very similar and there is evidence to justify this assumption. Dotted lines represent many levels of abstraction, i.e. the relationship between our formal models of programming languages, and real programming languages.

Chapter 4 defines a chorded language $\mathcal{L}^A$ with asynchronous and synchronous chords, and also a chorded language $\mathcal{L}^S$ with just synchronous chords and a `spawn` statement. Ultimately we relate $\mathcal{L}^S$ to the languages with monitors rather than $\mathcal{L}^A$ since $\mathcal{L}^S$ programs spawn threads in the same way as the languages with monitors, and this makes the translation of programs simpler. Chapter 5 defines a translation from $\mathcal{L}^A$ to $\mathcal{L}^S$, and proves the translation preserves program validity and structure. Chapter 6 defines the translation in the opposite direction, and again proves the preservation of program validity and structure.

Chapter 7 is the first time we consider monitors. We define a language with monitors, $\mathcal{L}^J$, and define a translation from $\mathcal{L}^J$ to a new chorded language $\mathcal{L}^{S+}$. The new language $\mathcal{L}^{S+}$ is just $\mathcal{L}^S$ extended with integers and fields. It would have been preferable to use $\mathcal{L}^S$ instead of $\mathcal{L}^{S+}$, but since the translation requires an integer "counter", it is best to include integers in the language. Also, while $\mathcal{L}^A$ and $\mathcal{L}^S$ can encode fields as proved in [4], this is not true of $\mathcal{L}^J$, so instead of encoding $\mathcal{L}^J$'s fields with chords, we leave this step out of the translation by including fields in $\mathcal{L}^{S+}$. Thus we give a translation from $\mathcal{L}^J$ into $\mathcal{L}^{S+}$. The preservation of program validity and structure over this translation is proven as before.

Chapter 8 concerns the opposite direction. We define an appropriate translation from $\mathcal{L}^S$ to a new language with monitors, $\mathcal{L}^{J*}$. Again, implementing the semantics of $\mathcal{L}^S$ requires more constructs than available in $\mathcal{L}^J$. The new language $\mathcal{L}^{J*}$ is a version of $\mathcal{L}^J$ with constructs that represent some implementation of a nondeterministic "queue" library. The preservation of program validity and structure over this translation is proven.

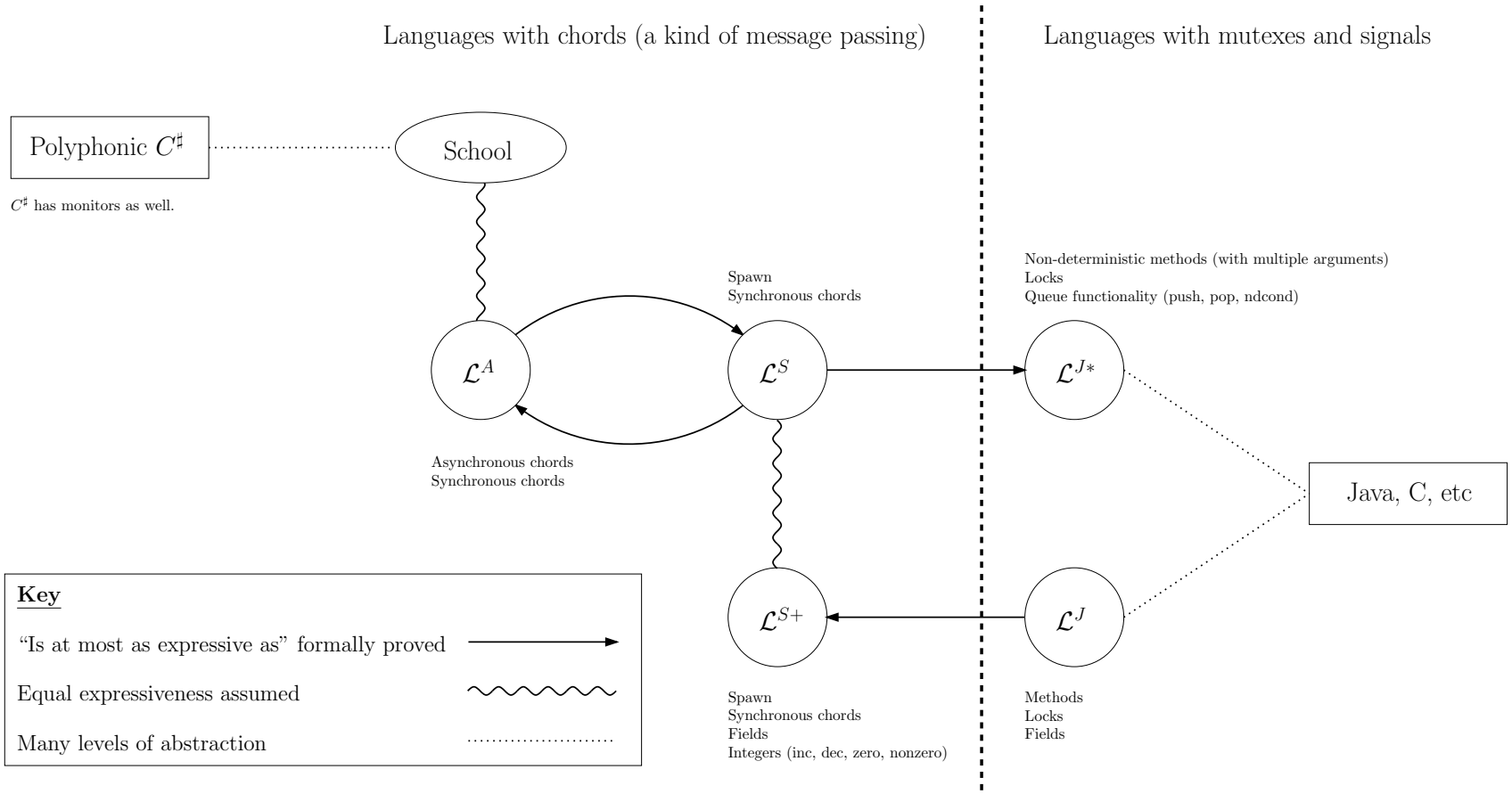
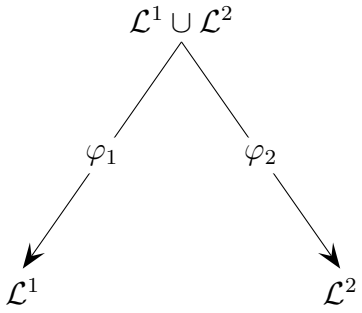


Figure 3.1: Overview of project

3.1 Extensions of [5]

3.1.1 When languages are unrelated by ($\subset$)

For languages where one language is not a restriction of the other, [5] suggests that we consider a language universe containing both languages, and try to show the eliminability of the set of constructs that distinguish the two languages. This is illustrated below:



Let:

- $\mathcal{L}^1$ and $\mathcal{L}^2$ be distinct languages and $\mathcal{L}^1 \cup \mathcal{L}^2$ be the language defined by the constructs in both $\mathcal{L}^1$ and $\mathcal{L}^2$.
- $\varphi_i : \mathcal{L}^1 \cup \mathcal{L}^2 \to \mathcal{L}^i$ (for $i \in \{1, 2\}$) be translations that preserve program validity, structure, and behaviour.

Let $\mathcal{L}^1 \preceq \mathcal{L}^2$ if and only if $\mathcal{L}^1$ is at most as expressive as $\mathcal{L}^2$. Now we can say that for all $i \in \{1, 2\}$, $\mathcal{L}^i \succeq \mathcal{L}^1 \cup \mathcal{L}^2$, due to φ_i. The reverse is due to the identity function, so $\mathcal{L}^i$ is as expressive as $\mathcal{L}^1 \cup \mathcal{L}^2$, and it follows that $\mathcal{L}^1$ is as expressive as $\mathcal{L}^2$.

In addition to [5], we note that φ_1 must be homomorphic over all constructs in $\mathcal{L}^1$ and thus over all constructs of $\mathcal{L}^1 \backslash \mathcal{L}^2$. Let $\mu_1 : \mathcal{L}^2 \to \mathcal{L}^1$ preserve program validity, structure (in terms of [5]), and behaviour, i.e. $\mu_1(c(e)) = c(\mu_1(e))$ for all constructs c in $\mathcal{L}^1 \cap \mathcal{L}^2$. We can use μ_1 to derive a suitable φ_1:

$$\varphi_1(c(e)) = \begin{cases} \mu_1(c(\varphi_1(e))) & \text{if } c \text{ in } \mathcal{L}^2 \\ c(\varphi_1(e)) & \text{if } c \text{ in } \mathcal{L}^1 \backslash \mathcal{L}^2 \end{cases}$$

Likewise we can define μ_1 in terms of φ_1 by restricting the domain of φ_1 to just $\mathcal{L}^2$. We can conclude that we need only specify a pair of appropriate translations $\mu_1 : \mathcal{L}^2 \to \mathcal{L}^1$ and $\mu_2 : \mathcal{L}^1 \to \mathcal{L}^2$ to show that $\mathcal{L}^1$ and $\mathcal{L}^2$ are equally expressive, in terms of [5]. Since these translations are smaller than φ_1 and φ_2, this is what we use in the rest of this report.

3.1.2 When languages are not homogeneous

The example languages in [5] (such as variations of the lambda calculus) were entirely described by a syntax definition of the form $e ::= ...|...|...$, we say they are homogeneous since every part of a program is defined by syntax. Our object-oriented formalisms have a more complex structure. The structure of programs includes not only the expressions that form message bodies (which are homogeneous), but also functions that define the set of classes, class inheritance, and the set of methods and method signatures within those classes.

The notion of structure-preservation defined in [5] (that translations must be homomorphic over shared program constructs and be macro translations of eliminated constructs) is ideal for our method *bodies*. This is because our method bodies are formalised by a homogeneous syntax of "source expressions" just like the lambda calculus in [5].

However, our formalism of programs has another layer of structure for which the technique does not readily apply. We now extend the definition of structure preservation so we can ensure our translated programs have equivalent program designs at this level as well.

If the languages we are comparing are identical at the level of method and class definitions, then there is no reason for the translation to interfere with this part of the structure. But what do we consider to be interference? How much can we change the top-level structure of a program before we change the program design? This is unfortunately rather subjective.

If methods or classes had to be removed, this would damage the modularity of the software so we would have to consider the program designs to be distinct, but what about adding methods or classes? As far as readability is concerned, methods and classes tend to be independent from other methods and other classes. I.e. one can read a class or method without having to read any other classes or methods. The addition of a method therefore does not interfere with the readability of the rest of the code.

Since the synchronisation behaviour of a chorded language is coupled with method invocation, it is inevitable that we will need to create new methods to implement synchronisation, when translating to chorded languages.

If the languages we are comparing are different at the level of method and class definitions, then we try to use the fact that our formalisations are just an alternative representation of the syntax that would realistically describe the structure of classes and methods. For example, when removing asynchronous chords from a language, the asynchronous chords lie alongside the synchronous chords, and are enclosed by the class definition. Thus the class definition and the synchronous chords should not be changed, but we will need to replace the asynchronous chords with something – more synchronous chords.

Although the structure of classes has changed, the expression syntax is the same, so because the translation must be homomorphic over all expression constructs, the chord bodies must not be changed. For the synchronous chords that have not changed, the bodies must be preserved intact. For the asynchronous chords that have somehow been simulated with synchronous chords, their bodies must be preserved in some way, inside whatever code is used to simulate the behaviour of asynchronous chords.

The situation with type signatures is more complicated, since type signatures have a wider-reaching effect than just the design of the class – they affect the validity of code in all parts of the program. Thus any translation that preserves the syntax of method calls, must preserve the type signatures of classes, although it can still add new methods and thus new signatures.

3.1.3 When code is duplicated

As an extra criterion for the requirements of structure preservation, we consider it beneficial that a translation should not duplicate any code from the original program. If a program design cannot be ported to the target language without duplicating code (and the resultant difficulties in maintaining said code), programmers will prefer to use the original language.

For example, in [5] there was an encoding of the `let` construct in the lambda calculus, using a substitution: $\varphi(\texttt{let}\ x = M\ \texttt{in}\ N) = N[M/x]$. This was structure-preserving according to [5], but since it has the potential to duplicate the code M in the generated program, we consider this a distinct program structure.

The translations in this report do not duplicate any parts of the original programs.

3.1.4 Exceptions to structure preservation

There are some cases where it is impossible to meet the requirements of structure preservation in our translations. This is because our formalisms are not perfectly accurate models of real programming languages, so we exempt these cases from the requirements.

Firstly, arguments are not identified by some defined name in our formalisations, as they are in real programming languages. For those formalisations where only one argument is allowed per invoked body, this is referred to with just `x`. Where there is one argument for each method that constitutes a chord, we identify the argument by the method that it came from, e.g. for a method m we use $m\_\texttt{x}$. Where there are many arguments for each method, we assume some specific ordering and identify the arguments with an index x_i. This means that when a method or chord body is moved into a different environment, the argument variables must be substituted throughout the expression, so that they refer to the right source of data. Argument variables are, however, a construct shared by all of our languages, so strictly they must be preserved. This is not a feature of the languages we are modelling, but a feature of our minimalist model of them, so we justify the substitution of argument variables as an exception to the structure preservation property. We could alternatively have solved this problem by extending our formalisms to be more realistic, and included argument identifiers, but we prefer to keep the semantics as simple and clear as possible.

Secondly, where a translated program needs to initialise its objects in a way in which the original program did not, we have to translate the `new` c expression to a `new` $c.init(\texttt{voidval})$ expression, where *init* is a method of some form that does the initialisation. Once again this is a feature of our formalisations rather than real languages, since any real object-oriented language has constructors. Constructors increase the expressiveness of a language because without them, you have to initialise every object at the point of instantiation as just shown. Instead of burdening our semantics with rules for constructors, we instead make an exception in the rules of structure preservation so that these translations can be used.

Chapter 4

Definition of chorded languages $\mathcal{L}^A$ and $\mathcal{L}^S$

Before we can formally prove properties of various kinds of programming languages, we must formally define those languages. We present two languages: $\mathcal{L}^A$, which has asynchronous chords for spawning new threads, and $\mathcal{L}^S$, which has a `spawn` statement for this purpose.

Asynchronous chords were described in section 2.3. The meaning of the statement `spawn` e is to invoke the execution of the block of code e in a new thread. In subsequent computation steps, the new thread will interleave with the other threads in the usual way. There is no "join" behaviour prescribed by the semantics, but such behaviour can be implemented by the programmer using synchronous chords. In other words, the new thread need not interact with the original thread at any time during its life.

We model the semantics of programs using small-step operational semantics (as is standard with concurrent languages), and we also model a well-formedness judgement which identifies a class of programs as being valid (using type systems to validate the source expressions). Although we have not proven "progress" of well-formed programs, we believe the well-formedness judgement excludes a large proportion of programs that can enter a run-time state that we consider to be malformed, and for which no execution step is defined.

Our aim is to show that programs that use asynchronous chords can instead use `spawn` without changing their design (chapter 5) and likewise programs using `spawn` can instead use asynchronous chords (chapter 6).

4.1 Comparison with the formalisations in [4]

The languages defined in this chapter ($\mathcal{L}^A$ and $\mathcal{L}^S$) are based on a formalisation of a chorded language called School, presented in [4]. School's type system was also shown to have the subject reduction property. $\mathcal{L}^A$ is more similar to School than $\mathcal{L}^S$, because as already stated, $\mathcal{L}^S$ does not have asynchronous chords and has a `spawn` construct instead.

Our language $\mathcal{L}^A$ takes from School the structure of programs and semantics for dealing with chord invocation and method calls (i.e. dealing with the queues). As in [4], we do not include field members in the formalisation since they can be represented with chords.

Aside from superficial technical differences, $\mathcal{L}^A$ differs from school because we remove the distinction between asynchronous and `void` methods that was present in the type system of [4]. In $\mathcal{L}^A$ we consider asynchronous methods to have return type `void`, whereas in [4] there was a special type `async`. This meant the set of asynchronous methods and synchronous methods was disjoint in School, since all asynchronous methods had return type `async` whereas synchronous methods had either return type `void` or c. For the programmer, this meant that a method could not be both a synchronous and asynchronous method, or in other words it could not be the synchronous part of a chord, at the same time as being an asynchronous part of a chord.

When trying to write the translation φ_{AS} (chapter 5), it was difficult to create a translation without needing to create a method that was both synchronous and asynchronous, and thus dropped the distinction in the type system. There were many places in the semantics of School where `async` and `void` were explicitly defined to subsume each other (i.e. be equivalent types) so dropping the distinction in $\mathcal{L}^A$ makes little difference to the proofs in [4]. We believe that return value is the only relevant concern in the type system, and since asynchronous methods return `voidval` when called, their return type should be `void`.

However, there are some strange programs that are now allowed by the well-formedness judgement. We can have a synchronous chord whose synchronous method is part of the chord as an asynchronous method as well:

```
void m() & async m() {
        ...
}
```

The above is a well-formed and well-behaved $\mathcal{L}^A$ and $\mathcal{L}^S$ program. The behaviour of calls to `m()` are non-deterministic, but the calls will always progress. If an object is available in the queue of `m()` then the chord body may be invoked, or the argument of the call may be queued with the others. If the queue is empty, the behaviour is always to add the argument to the queue. Note that because the argument type is `void` here, the queue does not contain any information beyond the number of members, so could be implemented with just a counter.

We believe from our study of potential translations, but we have not formally proven, that $\mathcal{L}^A$ is equivalent to the language School in terms of expressiveness. It is possible to express the above program in School as the following example shows:

```
void m() & async m'() {
        ...
}
void m() {
        m'();
}
```

We did not want to include this extra step in our translation, however, so we defined $\mathcal{L}^A$ differently to School.

4.2 Guided tour of $\mathcal{L}^A$ and $\mathcal{L}^S$

The syntax and semantics of $\mathcal{L}^A$ and $\mathcal{L}^S$ are given in figure 4.1 and figure 4.2 respectively. The well-formedness judgement is the same for both languages and is specified in figure 4.3. First we look at the syntax and semantics of $\mathcal{L}^A$ in detail, then we summarise the differences between $\mathcal{L}^A$ and $\mathcal{L}^S$, which are very slight. Finally we will consider the well-formedness judgement (that applies to both languages).

Many of the atoms we use to denotes various aspects of programs, are shared between the languages. For instance a program is denoted P in both languages. Where it may be confusing what language the program is written in, we decorate with the name of the language, e.g. P^A or P^S. Likewise, some of the functions are overloaded, and are defined differently depending on whether their arguments are a part one language or the other. For instance $\mathcal{S}up(P^A, c)$ is defined to be $P^A{\downarrow_4}$ whereas $\mathcal{S}up(P^S, c)$ is defined to be $P^S{\downarrow_3}$. In these cases, the decoration is always given. These higher-level definitions act like a "portability layer". They allow us to think in terms of the more abstract concept of superclass, without needing to consider what element of the program tuple represents this information.

4.2.1 Programs

$\mathcal{L}^A$ programs, e.g. P^A, are tuples of 4 functions. Those functions represent the type annotations, synchronous and asynchronous chords in each class, and also the class that each class inherits.

The only type annotations in the program are the method signatures, i.e. the return type and argument type for each method. We need only support one argument in each method because it is easy to simulate multiple arguments by using a chord with one method for each argument. When no argument is required, we can use a type of `void`. The method signature is the first element of the program tuple, but we usually refer to it with the auxiliary notation $\mathcal{M}(P, c, m)$.

As explained in section 2.3, synchronous chords have one synchronous method, and $n \geq 0$ asynchronous methods. Because two chords can have the same synchronous method, a method can be the synchronous part of many chords. We represent the set of synchronous chords, of which a method m is the synchronous part, with $\mathcal{SC}hs(P, c, m)$. This returns a set of chords, each of which is a tuple containing the set of asynchronous methods and a method body.

Asynchronous chords are comprised entirely of asynchronous methods. No method is special in the set of asynchronous methods that comprises an asynchronous chord, so we just collect the set of asynchronous chords and represent it with $\mathcal{AC}hs(P, c, m)$.

Every class in the program extends another class. The set of classes in a program, is in fact the set of classes c for which $\mathcal{S}up(P, c)$ is defined. There is a class $\texttt{Object} \in Id_c$ that has no members, for which $\mathcal{S}up(P, \texttt{Object})$ is not defined. Thus this class is not part of any program, but other objects can extend it, if they do not extend any of the classes within the program.

Source expressions are used in chord bodies, and are defined by a context free grammar, as is standard. The atom `null` represents the lack of an object, whereas `voidval` represents the lack of any value at all. `this` refers to the value of the current object's

address, and $m\_\texttt{x}$ is the argument variable that was supplied by the method m of the enclosing chord. We refer to argument variables by the name of the method which supplies them. `new` c instantiates a class and returns the address of the instance, and the other two constructs – method call and sequential composition are standard in object oriented imperative languages.

The function $\mathcal{Q}(P, c)$ is the set of queues required for a class c in program P. As described in section 2.3, the arguments of asynchronous method calls are placed on an unordered queue. Thus for each asynchronous method in a class, we need a queue. The definition of $\mathcal{Q}(P, c)$ therefore extracts the asynchronous methods from both the synchronous and asynchronous chords.

4.2.2 Execution

Execution is modelled with a state transition relation. Using a relation instead of a function allows us to model the intrinsic non-determinism of concurrent programming languages. Each run-time state can transition to one of a set of run-time states. The set of possible transitions depends not only on the current state, but also on the program being executed.

The state of execution consists of some threads and a shared heap. The state of a thread is modelled with run-time expressions that mutate slightly with each step of execution. Real implementations would use a stack, and a program counter, but it is simpler for us to store this information in an expression with a well-defined syntax. This syntax is the same as that which defines source expressions, except the variable `this` and argument variables are replaced with concrete values such as addresses of instances on the heap.

The heap is represented with a partial function from the natural numbers to objects. Unallocated addresses are undefined in the mapping. Instances are represented using tuples that contain the instance's class (to facilitate late-binding), and the queues within that instance.

Since we have only one argument in our method calls, we need only store single values in the queues. Since the queue is an unordered collection of values, the set of queues is $Multiset(Val)$, which is the set of multisets of Val. A multiset of S is a sequence of members of S, e.g. $(s_1, \ldots, s_n)$ which we consider equivalent modulo re-ordering, i.e. all the permutations of a multiset are the same multiset. A multiset is not the same as a *set*, however, because a multiset can have duplicated elements whereas a set cannot.

The set of run-time states is $State = Multiset(RunExpr) \times Heap$, i.e. a run-time state consists of a multiset of threads and a shared heap. We therefore consider a sequence of threads, but we do not observe any kind of ordering on these threads. We could have expressed this with a semantics rule that permuted the threads, but I feel this is a technicality of our formalism rather than a feature of the languages we are modelling, so it does not belong explicitly in the set of semantics rules.

The relation that defines execution, $(\rightsquigarrow)$, is of type $\mathcal{L} \times State \times State$, but we denote instances of the relation as so: $P \vdash (\{e_1 \ldots e_n\}, h) \rightsquigarrow (\{e'_1 \ldots e'_m\}, h')$ because the program P guides the course of execution. The brackets do not resolve any ambiguity, so we may remove them to save space. We contract the relation to $P \vdash e_1 \ldots e_n, h \rightsquigarrow e'_1 \ldots e'_m, h'$. The rules that define this relation are (Run), (Strung) and (Spawn).

Because our semantics are a small-step operational semantics, we use a context syntax, to express the idea that only a small part of the run-time expression is relevant to the

execution step. Only the relevant part will be affected by the execution step, the rest will remain unchanged. A context of an expression represents all the parts of the expression that are inert because either they are waiting for something (such as the resolution of a sub-expression to a value) or they have already evaluated to a value and the focus of execution has moved to another part of the expression.

For example, consider the expression $\texttt{voidval}; \iota.m(\iota.m'(v)) = E[\iota.m'(v)]$ where $E[\cdot] = \texttt{voidval}; \iota.m(\cdot)$. The context $E[\cdot]$ allows us to package away and ignore the irrelevant parts of the run-time expression, when specifying the course of execution. It allows us to specify execution order for method calls and sequential composition, because we define what part of the expression is to be executed, and which is to be packaged away and ignored.

The rule (Run) introduces another kind of state transfer relation, that only considers a single thread. This is factoring out what would be duplication in our rules, because there are number of semantics rules that apply to a thread without reference to any other threads. These rules are (New), (InvA), and (InvS). It also abstracts away from the context of the thread, so the rules (New), (InvA), and (InvS) only apply to threads with no contexts.

The rule (New) specifies that execution of `new` c should define a new instance at some previously unallocated address on the heap, with an empty set of queues. We use the lambda notation to define the function that represents the instance's queues.

The rule (InvA) defines the execution of an asynchronous method call, i.e. putting the argument on the queue and returning immediately. We bind c to the instance ι's class, and qs to its queues (which is a mapping from asynchronous methods to multisets of values). Recall that a method of class c is asynchronous if and only if it is a member of $\mathcal{Q}(P, c)$, since $\mathcal{Q}$ is defined to extract the asynchronous methods from both synchronous and asynchronous chords. We use $\uplus$ to denote multiset union, which we use to represent the addition of the argument value to the queue.

The rule (InvS) defines the behaviour of synchronous method calls. This involves extraction of the various arguments from the various queues. The choice of which chord to invoke is arbitrary, so long as it is a synchronous chord of which m is the synchronous method, i.e. if it is in $\mathcal{SCHS}(P, c, m)$. We bind $m_1 \ldots m_n$ to the asynchronous methods in the chosen chord, and $\mathbf{e}$ as the body of the chosen chord. The body will be pushed into the run-time expression with an appropriate substitution to turn it from a source expression into a run-time expression, i.e. replacing references to argument variables and `this` with values. We extract the arguments from the qs by defining a new queue for each asynchronous method q_i, that is one member less than the original queue $qs(m_i)$. Thus the rule only applies when all the relevant queues are non-empty. The value removed is used in the substitution of argument variables, and the new queue is mapped over the top of the old one, in the resultant heap.

The rule (Strung) is very similar to (InvS) because it is also invoking a chord body and using an appropriate substitution to convert it into a run-time expression. The main difference is that it spawns a new run-time expression alongside the existing threads. Thus it does not require any of the existing threads to be in any particular state, and can "fire" at any time if the relevant queues are non-empty.

Execution of the program can be stuck, i.e. there are no rules that apply to the current run-time state, in a number of conditions. Null pointer de-referencing – there is no provision for any kind of null pointer exception in the event of trying to execute $\texttt{null}.m(\_)$.

Deadlock – all "active" threads blocked at a synchronous method call because some queues are non-empty and no threads are running so no values will be put into these queues. No chord defined – we can define a program that has a class with methods, but no chord defined that uses those methods (this means those methods are not in $\mathcal{Q}$, nor are there any synchronous chords associated with the method, so neither (InvA) not (InvS) apply). Valid termination – when execution has reduced the run-time expression to a value (in a context), the execution is "complete".

The semantics is non-deterministic for several independent reasons. An arbitrary thread is selected from the multiset in rule (Run). If a method is both an asynchronous method and a synchronous method, both (InvA) and (InvS) may be applicable. The (Strung) rule may apply at the same time as other rules apply. The selection of a value (in (InvS), (Strung)) from a queue that is greater than one element in size is non-deterministic because the queues are unordered. It is non-deterministic what address is used to store newly-constructed instances, but this is of very little consequence to the execution of the program, since we can consider the states equal up to renaming of addresses. If there are multiple chords whose queues are non-empty, the choice of which chord to invoke is arbitrary.

The difference between $\mathcal{L}^A$ and $\mathcal{L}^S$ is slight: With no asynchronous chords, the definition of $\mathcal{L}^S$ programs is a tuple of 3 elements instead of 4. The strung rule is unapplicable for no chord are in $\mathcal{ACHS}(P^S, c, m)$. The expressions (both source and run-time) syntax are extended with the `spawn` e construct. Finally there is a rule (Spawn) that describes how `spawn` e is executed. It puts the e into its own thread, and replaces the `spawn` statement with `voidval`.

4.2.3 Well-formed programs

A program is well-formed ($\vdash P$) if all its classes are well-formed. Recall that the set of classes in a program is the set of classes for which $\mathcal{Sup}(P, c)$ is defined, because every class has a parent class, but not every class has methods, or chords.

A class is well-formed if it extends a valid class (valid classes are the set of classes in the program, and the class `Object`). We also require that method type signatures be preserved over inheritance, although no inheritance of chords is required. As is standard, we allow subtypes to be returned, and super-types to be consumed as arguments of the methods in the subclass.

Unfortunately, while the inheritance of method bodies in conventional languages is straight forward, it is not obvious how to inherit the chords themselves, so we side-step this issue. One side-effect of this is that it is possible to extend a class but define no chords, in which case any method invocations will result in a stuck execution.

We require the type of chord bodies to match the return type of their synchronous methods (with synchronous chord) or to be `void` (for asynchronous chords). It is easy to make any expression **e** of type t, into a `void` type by using instead the expression **e**; `voidval`.

Programs:

$$
\begin{array}{rcll}
P^A \in \mathcal{L}^A & = & Id_c \times Id_m \to Methsig & \text{(Type signatures)} \\
& \times & Id_c \times Id_m \to \mathbb{P}(Chord^A) & \text{(Synchronous chords)} \\
& \times & Id_c \to \mathbb{P}(Chord^A) & \text{(Asynchronous chords)} \\
& \times & Id_c \to Id_c & \text{(Superclass)} \\
Methsig & ::= & t\ m(t) & \\
t \in Type & ::= & c \mid \texttt{void} & \\
Chord^A & = & \mathbb{P}(Id_m) \times SrcExpr^A & \\
\mathbf{e}^A \in SrcExpr^A & ::= & \texttt{null} \mid \texttt{voidval} \mid \texttt{this} \mid m\_\texttt{x} \mid \texttt{new}\ c \mid \mathbf{e}^A.m(\mathbf{e}^A) \mid \mathbf{e}^A\ ;\ \mathbf{e}^A & \\
m \in Id_m & & c \in Id_c &
\end{array}
$$

$$\mathcal{M}(P,c,m) = P{\downarrow_1}(c,m) \qquad \mathcal{SC}\mathit{hs}(P,c,m) = P{\downarrow_2}(c,m)$$

$$\mathcal{AC}\mathit{hs}(P^A,c) = P^A{\downarrow_3}(c) \qquad \mathcal{S}\mathit{up}(P^A,c) = P^A{\downarrow_4}(c)$$

$$\mathcal{Q}(P,c) = \bigcup\{\ ch{\downarrow_1} \mid ch \in \mathcal{AC}\mathit{hs}(P,c)\ \vee\ \exists m.ch \in \mathcal{SC}\mathit{hs}(P,c,m)\ \}$$

Runtime objects and semantics:

$$
\begin{array}{rcll}
h \in Heap & = & \mathbb{N} \to (Id_c \times (Id_m \to Multiset(Val))) & \\
e^A \in RunExpr^A & ::= & v \mid \texttt{new}\ c \mid e^A.m(e^A) \mid e^A\ ;\ e^A & \\
v \in Val & ::= & \texttt{null} \mid \texttt{voidval} \mid \iota & \\
\iota \in \mathbb{N} & & \text{(Addresses)} & \\
E^A[\cdot] & ::= & E^A[\cdot].m(e^A)\ \mid\ \iota.m(E^A[\cdot])\ \mid\ E^A[\cdot]\ ;\ e^A\ \mid\ v\ ;\ E^A[\cdot] &
\end{array}
$$

$$\frac{P \vdash e, h \rightsquigarrow e', h'}{P \vdash e_1 \ldots e_n, E[e], h \rightsquigarrow e_1 \ldots e_n, E[e'], h'} \quad \text{(Run)}$$

$$\frac{h(\iota) = \mathcal{U}\mathit{df}}{P \vdash \texttt{new}\ c, h \rightsquigarrow \iota, h[\iota \mapsto [\![c \| \lambda m.\emptyset]\!]]} \quad \text{(New)}$$

$$\frac{h(\iota) = [\![c \| qs]\!], \qquad m \in \mathcal{Q}(P,c)}{P \vdash \iota.m(v), h \rightsquigarrow \texttt{voidval}, h[\iota \mapsto [\![c \| qs[m \mapsto qs(m) \uplus \{v\}]]\!]]} \quad \text{(InvA)}$$

$$\frac{\begin{array}{c} h(\iota) = [\![c \| qs]\!], \qquad (\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{SC}\mathit{hs}(P,c,m) \\ \forall i \in \{1 \ldots n\}.qs(m_i) = \{v_i\} \uplus q_i \\ h' = h[\iota \mapsto [\![c \| qs[m_1 \mapsto q_1 \ldots m_n \mapsto q_n]]\!]] \end{array}}{P \vdash \iota.m(v), h \rightsquigarrow \mathbf{e}[v_1/m_1\_\texttt{x}\ \ldots\ v_n/m_n\_\texttt{x},\ v/m\_\texttt{x},\ \iota/\texttt{this}], h'} \quad \text{(InvS)}$$

$$\frac{\begin{array}{c} h(\iota) = [\![c \| qs]\!], \qquad (\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{AC}\mathit{hs}(P^A,c) \\ \forall i \in \{1 \ldots n\}.qs(m_i) = \{v_i\} \uplus q_i \\ h' = h[\iota \mapsto [\![c \| qs[m_1 \mapsto q_1 \ldots m_n \mapsto q_n]]\!]] \end{array}}{P^A \vdash e_1 \ldots e_n, h \rightsquigarrow e_1 \ldots e_n, \mathbf{e}[v_1/m_1\_\texttt{x}\ \ldots\ v_n/m_n\_\texttt{x},\ \iota/\texttt{this}], h'} \quad \text{(Strung)}$$

Figure 4.1: Syntax and semantics for $\mathcal{L}^A$.

Programs:

$$
\begin{array}{rcll}
P^S \in \mathcal{L}^S & = & Id_c \times Id_m \to Methsig & \text{(Type signatures)} \\
& \times & Id_c \times Id_m \to \mathbb{P}(Chord^S) & \text{(Synchronous chords)} \\
& \times & Id_c \to Id_c & \text{(Superclass)} \\
Methsig & ::= & t\ m(t) & \\
t \in Type & ::= & c \mid \texttt{void} & \\
Chord^S & = & \mathbb{P}(Id_m) \times SrcExpr^S & \\
\mathbf{e}^S \in SrcExpr^S & ::= & \texttt{null} \mid \texttt{voidval} \mid \texttt{this} \mid m\_\texttt{x} \mid \texttt{new}\ c \mid \mathbf{e}^S.m(\mathbf{e}^S) \mid \mathbf{e}^S\ ;\ \mathbf{e}^S & \\
& \mid & \texttt{spawn}\ \mathbf{e}^S & \\
m \in Id_m & & c \in Id_c &
\end{array}
$$

$$\mathcal{M}(P,c,m) = P{\downarrow_1}(c,m) \qquad \mathcal{SCHs}(P,c,m) = P{\downarrow_2}(c,m)$$

$$\mathcal{ACHs}(P^S,c) = \emptyset \qquad \mathcal{Sup}(P^S,c) = P^S{\downarrow_3}(c)$$

$$\mathcal{Q}(P,c) = \bigcup\{\ ch{\downarrow_1} \mid ch \in \mathcal{ACHs}(P,c)\ \vee\ \exists m.ch \in \mathcal{SCHs}(P,c,m)\ \}$$

Runtime objects and semantics:

$$
\begin{array}{rcl}
e^S \in RunExpr^S & ::= & v \mid \texttt{new}\ c \mid e^S.m(e^S) \mid e^S\ ;\ e^S \mid \texttt{spawn}\ e^S \\
E^S[\cdot] & ::= & E^S[\cdot].m(e^S)\ \mid\ \iota.m(E^S[\cdot])\ \mid\ E^S[\cdot]\ ;\ e^S\ \mid\ v\ ;\ E^S[\cdot]
\end{array}
$$

(Run) (New) (InvA) (InvS) from figure 4.1

$$\frac{}{P^S \vdash e_1 \ldots e_n, E[\texttt{spawn}\ e], h \rightsquigarrow e_1 \ldots e_n, E[\texttt{voidval}], e, h} \quad \text{(Spawn)}$$

Figure 4.2: Syntax and semantics for $\mathcal{L}^S$.

Well-formedness:

$$\frac{\forall c.\mathcal{S}up(P,c) \neq \mathcal{U}df \implies P \vdash c}{\vdash P} \quad \text{(WFProg)}$$

$$\frac{\begin{array}{l} P \vdash \mathcal{S}up(P,c) \diamond_{cl} \\ \forall m.\mathcal{M}(P,\mathcal{S}up(P,c),m) = t_r\ m(t_a) \implies \exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a.\mathcal{M}(P,c,m) = t'_r\ m(t'_a) \\ \forall m, (\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{SC}hs(P,c,m). \\ \qquad \forall i \in \{1 \ldots n\}.\mathcal{M}(P,c,m_i) = \texttt{void}\ m(t_i), \\ \qquad \mathcal{M}(P,c,m) = t\ m(t_0), \\ \qquad P, [m_1\_\texttt{x} \mapsto t_1 \ldots m_n\_\texttt{x} \mapsto t_n, m\_\texttt{x} \mapsto t_0, \texttt{this} \mapsto c] \vdash \mathbf{e} : t \\ \forall (\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{AC}hs(P,c). \\ \qquad \forall i \in \{1 \ldots n\}.\mathcal{M}(P,c,m_i) = \texttt{void}\ m(t_i), \\ \qquad P, [m_1\_\texttt{x} \mapsto t_1 \ldots m_n\_\texttt{x} \mapsto t_n, \texttt{this} \mapsto c] \vdash \mathbf{e} : \texttt{void} \end{array}}{P \vdash c} \quad \text{(WFClass)}$$

$$\frac{\mathcal{S}up(P,c) \neq \mathcal{U}df \quad \vee \quad c = \texttt{Object}}{P \vdash c \diamond_{cl}} \quad \text{(IsClass)}$$

Source type rules:

$$\frac{v \in \{\texttt{this}, m\_\texttt{x}\}}{P,\Gamma \vdash v : \Gamma(v)} \quad \text{(STVar)} \qquad \frac{P \vdash c \diamond_{cl}}{P,\Gamma \vdash \texttt{null} : c} \quad \text{(STNull)}$$

$$\frac{}{P,\Gamma \vdash \texttt{voidval} : \texttt{void}} \quad \text{(STVoid)} \qquad \frac{P \vdash c \diamond_{cl}}{P,\Gamma \vdash \texttt{new}\ c : c} \quad \text{(STNew)}$$

$$\frac{\begin{array}{c} P,\Gamma \vdash \mathbf{e}_1 : c \\ P,\Gamma \vdash \mathbf{e}_2 : t \\ \mathcal{M}(P,c,m) = t_r\ m(t) \end{array}}{P,\Gamma \vdash \mathbf{e}_1.m(\mathbf{e}_2) : t_r} \quad \text{(STInv)} \qquad \frac{\begin{array}{c} P,\Gamma \vdash \mathbf{e}_1 : t_1 \\ P,\Gamma \vdash \mathbf{e}_2 : t_2 \end{array}}{P,\Gamma \vdash \mathbf{e}_1\ ;\ \mathbf{e}_2 : t_2} \quad \text{(STSeq)}$$

$$\frac{\begin{array}{c} P,\Gamma \vdash \mathbf{e} : c \\ P,\Gamma \vdash \mathcal{S}up(P,c) : c' \end{array}}{P,\Gamma \vdash \mathbf{e} : c'} \quad \text{(STSub)} \qquad \frac{P,\Gamma \vdash \mathbf{e} : \texttt{void}}{P,\Gamma \vdash \texttt{spawn}\ \mathbf{e} : \texttt{void}} \quad \text{(STSpawn)}$$

We also define $c \sqsubseteq c'$ (in the context of some program P) as the reflexive transitive closure of the inheritance relation defined by $\mathcal{S}up$, i.e. $c \sqsubseteq c$, $c \sqsubseteq \mathcal{S}up(P,c)$, and $c \sqsubseteq c' \wedge c' \sqsubseteq c'' \implies c \sqsubseteq c''$.

Figure 4.3: Rules for well-formed chorded programs in $\mathcal{L}^A$ and $\mathcal{L}^S$.

Chapter 5

$\mathcal{L}^A$ is at most as expressive as $\mathcal{L}^S$

In this chapter we define a translation φ_{AS} from $\mathcal{L}^A$ to $\mathcal{L}^S$. We show that the translation preserves program structure, validity, and behaviour.

5.1 Definition of the translation φ_{AS}

5.1.1 Examples

Before giving the translation, we show how a fairly general asynchronous chord, can be converted into a pair of synchronous chords, using the `spawn` statement. Since each asynchronous chord and each class is treated independently in the translation, we can imagine how this process scales up to arbitrary programs. Below we give the example program as the programmer might write it, and as our formalism represents it.

```
class c {
        async m1(t1 m1_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
                e_async;
        }
}
```

$\mathcal{M}(P^A, c, m_1) = \texttt{void}\ m(t_1),\ \mathcal{M}(P^A, c, m_2) = \texttt{void}\ m(t_2)\ \ldots\ \mathcal{M}(P^A, c, m_n) = \texttt{void}\ m(t_n)$
$\forall m.\mathcal{SC}hs(P^A, c, m) = \emptyset$
$\mathcal{AC}hs(P^A, c) = \{(\{m_1 \ldots m_n\}, e_{async})\}$
$\mathcal{S}up(P^A, c) = \texttt{Object}$
(and thus we can derive $\mathcal{Q}(P^A, c) = \{m_1 \ldots m_n\}$)

If we choose a method from the above asynchronous chord, say `m1`, and change it from an `async` to a `void`, we will have a synchronous chord instead of an asynchronous one. However, now if `m1` is invoked when one of the queues for the other methods is empty, the calling thread will be blocked, so the behaviour is not the same.

To remedy this, we use the `spawn` construct to allow this blocking to occur in a new thread, and the calling thread can proceed as before. We could wrap every invocation of `m1` with `spawn m1` throughout the program, but this would not preserve the program structure. Instead, we remove the chosen method from the chord, leaving in its place a "new" synchronous method `m1'`. We call this process "plucking", and the new method is

called the "shadow" of the plucked method, so here we plucked `m1` from the chord, and `m1'` is its shadow. The shadow will not interfere with any other chord joins since it is "new" and therefore not part of any other chords.

The plucked method acts as an asynchronous wrapper around the shadow. It uses the `spawn` construct, to allow the blocking (that calling the shadow causes when a queue is empty) to occur in a new thread.

```
class c {
        void m1(t1 m1_x) {
                spawn this.m1'(m1_x)
        }

        void m1'(t1 m1'_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
                e_async[m1'_x/m1_x];
        }
}
```

$\mathcal{M}(P^S, c, m_1) = \texttt{void}\ m(t_1),\ \mathcal{M}(P^S, c, m_2) = \texttt{void}\ m(t_2)\ \ldots\ \mathcal{M}(P^S, c, m_n) = \texttt{void}\ m(t_n)$
$\mathcal{M}(P^S, c, m_1') = \texttt{void}\ m(t_1)$
$\mathcal{ACHS}(P^S, c) = \emptyset$
$\mathcal{SCHS}(P^S, c, m_1) = \{(\emptyset, \texttt{spawn this.}m_1'(m_1\texttt{\_x}))\}$
$\mathcal{SCHS}(P^S, c, m_1') = \{(\{m_2 \ldots m_n\}, e_{async}[m_1'\texttt{\_x}/m_1\texttt{\_x}])\}$
$\mathcal{SUP}(P^S, c) = \texttt{Object}$
(and thus we can derive $\mathcal{Q}(P^A, c) = \{m_1 \ldots m_n\}$)

Calls to the plucked method have the same behaviour as before. The newly-spawned thread will act as a queue, waiting for the queues of `m2` ... `mn` to become non-empty before proceeding with the body `e_async`. Note that we have not changed the `e_async` code (up to renaming of argument variables). No other code should call the shadow method.

As another example, we create a second chord with the same method 'footprint' but with a different body:

```
class c {
        async m1(t1 m1_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
                e_async;
        }
        async m1(t1 m1_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
                e_async';
        }
}
```

$\mathcal{M}(P^A, c, m_1) = \texttt{void}\ m(t_1),\ \mathcal{M}(P^A, c, m_2) = \texttt{void}\ m(t_2)\ \ldots\ \mathcal{M}(P^A, c, m_n) = \texttt{void}\ m(t_n)$
$\forall m. \mathcal{SCHS}(P^A, c, m) = \emptyset$
$\mathcal{ACHS}(P^A, c) = \{(\{m_1 \ldots m_n\}, e_{async}), (\{m_1 \ldots m_n\}, e'_{async})\}$
$\mathcal{SUP}(P^A, c) = \texttt{Object}$
(and thus we can derive $\mathcal{Q}(P^A, c) = \{m_1 \ldots m_n\}$)

When we translate this program, we must encode both asynchronous chords. We can either pluck the same method from both or pluck a different method in each. The behaviour is the same either way. In the translation given in this chapter, we must specify a different wrapper method for each, regardless of whether we are plucking the same method. Here is the translated code where we pluck the same method:

```
class c {
        void m1(t1 m1_x) {
                spawn this.m1'(m1_x)
        }
        void m1(t1 m1_x) {
                spawn this.m1''(m1_x)
        }
        void m1'(t1 m1'_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
                e_async[m1'_x/m1_x];
        }
        void m1''(t1 m1''_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
                e_async'[m1''_x/m1_x];
        }
}
```

$\mathcal{M}(P^S, c, m_1) = \texttt{void}\ m(t_1),\ \mathcal{M}(P^S, c, m_2) = \texttt{void}\ m(t_2)\ \ldots\ \mathcal{M}(P^S, c, m_n) = \texttt{void}\ m(t_n)$
$\mathcal{M}(P^S, c, m_1') = \texttt{void}\ m(t_1)$
$\mathcal{M}(P^S, c, m_1'') = \texttt{void}\ m(t_1)$
$\mathcal{AChs}(P^S, c) = \emptyset$
$\mathcal{SChs}(P^S, c, m_1) = \{(\emptyset, \texttt{spawn this.}m_1'(m_1\texttt{\_x})), (\emptyset, \texttt{spawn this.}m_1''(m_1\texttt{\_x}))\}$
$\mathcal{SChs}(P^S, c, m_1') = \{(\{m_2 \ldots m_n\}, e_{async}[m_1'\texttt{\_x}/m_1\texttt{\_x}])\}$
$\mathcal{SChs}(P^S, c, m_1'') = \{(\{m_2 \ldots m_n\}, e'_{async}[m_1''\texttt{\_x}/m_1\texttt{\_x}])\}$
$\mathcal{Sup}(P^S, c) = \texttt{Object}$
(and thus we can derive $\mathcal{Q}(P^A, c) = \{m_1 \ldots m_n\}$)

Here, we pluck `m1` from the first chord and `m2` from the second:

```
class c {
        void m1(t1 m1_x) {
                spawn this.m1'(m1_x)
        }
        void m2(t2 m2_x) {
                spawn this.m2'(m1_x)
        }
        void m1'(t1 m1'_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
                e_async[m1'_x/m1_x];
        }
        async m1(t1 m1_x) & void m2'(t2 m2'_x) & ... & async mn(tn mn_x) {
                e_async'[m2'_x/m2_x];
        }
}
```

$\mathcal{M}(P^S, c, m_1) = \texttt{void}\ m(t_1),\ \mathcal{M}(P^S, c, m_2) = \texttt{void}\ m(t_2)\ \ldots\ \mathcal{M}(P^S, c, m_n) = \texttt{void}\ m(t_n)$
$\mathcal{M}(P^S, c, m_1') = \texttt{void}\ m(t_1)$
$\mathcal{M}(P^S, c, m_2') = \texttt{void}\ m(t_2)$
$\mathcal{AChs}(P^S, c) = \emptyset$
$\mathcal{SChs}(P^S, c, m_1) = \{(\emptyset, \texttt{spawn this.}m_1'(m_1\texttt{\_x}))\}$
$\mathcal{SChs}(P^S, c, m_2) = \{(\emptyset, \texttt{spawn this.}m_2'(m_2\texttt{\_x}))\}$
$\mathcal{SChs}(P^S, c, m_1') = \{(\{m_2 \ldots m_n\}, e_{async}[m_1'\texttt{\_x}/m_1\texttt{\_x}])\}$
$\mathcal{SChs}(P^S, c, m_2') = \{(\{m_1, m_3 \ldots m_n\}, e'_{async}[m_2'\texttt{\_x}/m_2\texttt{\_x}])\}$
$\mathcal{Sup}(P^S, c) = \texttt{Object}$
(and thus we can derive $\mathcal{Q}(P^A, c) = \{m_1 \ldots m_n\}$)

Note that in the above example, m_1 and m_2 are both synchronous and asynchronous methods in different chords. With this example, we can avoid this situation by plucking the same method from both chords, but this will not always be possible:

```
class c {
        async m2(t2 m2_x) {
                e_a;
        }
        void m1(t1 m1_x) & async m2(t2 m2_x) {
                e_s;
        }
}
```

The above program can only translate to the following code, since we have no choice but to pluck `m2` from the asynchronous chord:

```
class c {
        void m2(t2 m2_x) {
                spawn this.m2'(m2_x);
        }
        void m2'(t2 m2'_x) {
                e_a;
        }
        void m1(t1 m1_x) & async m2(t2 m2_x){
                e_s;
        }
}
```

This is why we changed the well-formedness relation of School – to allow these translations to be well-formed.

5.1.2 Translation

The encoding is given in figure 5.1 (recall that $\sqsubseteq$ is the reflexive transitive closure of the inheritance relation, defined in figure 4.3). Because the encoding defines a range of equivalent translated programs (since the choice of which method to pluck, as well as the choice of name for the shadow method, are arbitrary), we collect this flexibility into a single "decision" function δ, which is provided as an argument to φ_{AS}. We can thus ensure that the translation produces valid programs, if we require some sensible properties from the decision function.

$$\delta : \Delta = (Id_c \times Chord^A) \to (Id_m \times Id_m)$$

For each asynchronous chord ch in class c, $\delta(c, ch)$ tells us which method is being plucked, and what method is the shadow. The result of $\varphi_{AS}(P^A, \delta)$ is only well-defined when δ is well-formed, i.e. that $P^A \vdash \delta$ as defined below:

$\varphi_{AS}(P^A, \delta) = P^S$ if and only if

$$\begin{array}{lcl} \mathcal{SChs}(P^S, c, m) & = & \mathcal{SChs}(P^A, c, m) \\ & \cup & \{\ (\ ch{\downarrow_1}\backslash\{m_c\},\ ch{\downarrow_2}[\texttt{m\_x}/m_c\texttt{\_x}]) \mid ch \in \mathcal{ACh s}(P^A, c), \delta(c, ch) = (m_c, m)\ \} \\ & \cup & \{\ (\emptyset, \texttt{spawn this.}m'(m\texttt{\_x}))\mid ch \in \mathcal{ACh s}(P^A, c), \delta(c, ch) = (m, m')\ \} \end{array}$$

$$\mathcal{Sup}(P^S, c) = \mathcal{Sup}(P^A, c)$$

$$\mathcal{M}(P^S, c, m) = \begin{cases} \mathcal{M}(P^A, c, m) & \text{if } \mathcal{M}(P^A, c, m) \neq \mathcal{Udf} \\ \texttt{void } m(t) & \text{if } \exists ch, c \sqsubseteq c'\ .\ \delta(c', ch) = (m_c, m), \mathcal{M}(P^A, c', m_c) = \texttt{void } m_c(t) \end{cases}$$

Figure 5.1: Translation $\varphi_{AS} : (\mathcal{L}^A \times \Delta) \to \mathcal{L}^S$

$$\begin{array}{lcl} P^A \vdash \delta & \iff & \forall c, ch \in \mathcal{ACh s}(P^A, c)\ . \\ & & \quad \delta(c, ch){\downarrow_1} \in ch{\downarrow_1}, \\ & & \quad \forall c', ch' \in \mathcal{ACh s}(P^A, c')\ .\ \delta(c', ch'){\downarrow_2} = \delta(c, ch){\downarrow_2} \Longrightarrow c' = c,\ ch' = ch \\ & & \quad \forall c'\ .\ \mathcal{M}(P^A, c', \delta(c, ch){\downarrow_2}) = \mathcal{Udf} \\ & & \forall c, ch.\delta(c, ch) \neq \mathcal{Udf} \Longrightarrow ch \in \mathcal{ACh s}(P^A, c) \end{array}$$

This only requires that the first element of any $\delta(c, ch)$ is a valid asynchronous method to alter, and that the new method is not only unique to that particular class and chord, but has not been already defined anywhere in the original program (including other classes). One might imagine that the new methods need only be distinct within each class, since the classes themselves are a kind of name space, so method names in different classes do not clash. As it turns out, they do clash because classes can inherit other classes, and thus a new method name in a base class can interfere with a new method name in a sub class. It seems simplest for us to ensure all new method names are distinct regardless of class boundaries.

The values of δ for the four translations given in the previous section are as follows:

$\delta = [(c, (\{m_1 \ldots m_n\}, e_{async})) \mapsto (m_1, m_1')]$
$\delta = [(c, (\{m_1 \ldots m_n\}, e_{async})) \mapsto (m_1, m_1'), (c, (\{m_1 \ldots m_n\}, e'_{async})) \mapsto (m_1, m_1'')]$
$\delta = [(c, (\{m_1 \ldots m_n\}, e_{async})) \mapsto (m_1, m_1'), (c, (\{m_1 \ldots m_n\}, e'_{async})) \mapsto (m_2, m_2')]$
$\delta = [(c, (\{m_2\}, e_a)) \mapsto (m_2, m_2')]$

5.1.3 Properties of the translation

These properties are not interesting results in the broader scope of this report, but are presented here because they reflect the characteristics of this translation, and thus help us understand it. If $P^S = \varphi_{AS}(P^A, \delta)$:

- $\forall c\ .\ \mathcal{Q}(P^A, c) \supseteq \mathcal{Q}(P^S, c)$ No queues are added over the translation, but some may be lost because we are representing the queues with idle threads, using `spawn`.
- From $P^A \vdash \delta$, we know that the definition of $\mathcal{M}(P^S, c, m)$ is unambiguous, although it is possible that none of the cases apply. If $c \sqsubseteq c'$ and $\delta(c', ch) = (m_c, m)$, then we

know that for all c'', $\mathcal{M}(P^A, c'', m) = \mathcal{U}df$. Likewise we know that no other $ch, c' \sqsupseteq c$ exist where $\delta(ch, c')$ defines m as the shadow message, so there is no ambiguity *within* the second case.

5.2 Preservation of structure (proof)

We first formally define the property of structure preservation for our programs P^S and P^A where $P^S = \varphi_{AS}(P^A, \delta)$. We refer to section 3.1.2, where the contribution of [5] is put into the context of our object-oriented formalisations. In order for the translation to be structure-preserving, it must satisfy:

- Every syntax construct for source expressions of $\mathcal{L}^A$ is also a construct of source expressions of $\mathcal{L}^S$, so source expressions must not be changed in the translation. This means that the method signatures must remain intact as otherwise it would not be possible to prove well-formedness of P^S.

  The bodies of synchronous and asynchronous chords are not modified in the translation, up to the renaming of arguments which we explicitly exempt from the structure-preservation requirements in section 3.1.4. □

- Method signatures are common to $\mathcal{L}^A$ and $\mathcal{L}^S$, so they must not be altered, although we do allow addition of new methods.
  $\forall c, m. \mathcal{M}(P^A, c, m) = msig \Rightarrow \mathcal{M}(P^S, c, m) = msig$

  From $P^A \vdash \delta$ we know that the two cases for the definition of $\mathcal{M}(P^S, c, m)$ in φ_{AS} are never both true, so when $\mathcal{M}(P^A, c, m) = msig$ i.e. $\mathcal{M}(P^A, c, m) \neq \mathcal{U}df$, we know that $\mathcal{M}(P^S, c, m) = \mathcal{M}(P^A, c, m) = msig$. □

- Synchronous chords are common to $\mathcal{L}^A$ and $\mathcal{L}^S$ so the synchronous chords in a class must be preserved, although new synchronous chords can be added:
  $\forall c, m.\ \mathcal{SCh}s(P^A, c, m) \subseteq \mathcal{SCh}s(P^S, c, m)$

  This can be seen from the translation because $ch \in \mathcal{SCh}s(P^A, c, m) \implies ch \in \mathcal{SCh}s(P^A, c, m) \cup \ldots \implies ch \in \mathcal{SCh}s(P^S, c, m)$. □

- Asynchronous chords are present in $\mathcal{L}^A$ but not $\mathcal{L}^S$, so the translation must create a macro encoding for each asynchronous chord. This means that there must be something used in place of an asynchronous chord, who's structure is not dependent on the asynchronous chord's body except that it contains one exact copy of the asynchronous chord's body.

  The translation creates a new synchronous chord containing the body of the asynchronous chord, which is acceptable, and also another synchronous chord containing an expression that is defined independently of the body. □

- The feature of classes is common to $\mathcal{L}^A$ and $\mathcal{L}^S$, so the structure of classes must be preserved, although we allow the addition of new classes:
  $\forall c. \mathcal{S}up(P^A, c) = c' \implies \mathcal{S}up(P^S, c) = c'$

  We can prove this by inspection of φ_{AS}, which actually has the stronger property that no classes are added or removed. □

5.3 Preservation of well-formedness (proof)

We need to establish that the well-formedness of the translated source code is preserved over translation, this is the notion of programness in [5]. Let $P^S = \varphi_{AS}(P^A, \delta)$. We must show that $\vdash P^A \Longrightarrow \vdash P^S$.

Lemma 5.3.1 $P^A \vdash c \diamond_{cl} \Longrightarrow P^S \vdash c \diamond_{cl}$

Proof:

$$\begin{array}{lcll} P^A \vdash c \diamond_{cl} & \Longrightarrow & \mathcal{S}up(P^A, c) \neq \mathcal{U}df \vee c = \texttt{Object} & \text{(IsClass)} \\ & \Longrightarrow & \mathcal{S}up(P^S, c) \neq \mathcal{U}df \vee c = \texttt{Object} & \text{(Def } \varphi_{AS}\text{)} \\ & \Longrightarrow & P^S \vdash c \diamond_{cl} & \text{(IsClass)} \end{array}$$

Lemma 5.3.2 *Method signatures are preserved over inheritance:*
If $\vdash P^A$ *and* $P^A = \varphi_{SA}(P^S, \delta)$ *then*
$\forall m.\mathcal{M}(P^S, \mathcal{S}up(P^S, c), m) = t_r\ m(t_a) \Longrightarrow \exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a.\mathcal{M}(P^S, c, m) = t'_r\ m(t'_a)$

Proof:
Let $\vdash P^A$ and $\mathcal{M}(P^S, \mathcal{S}up(P^S, c), m) = t_r\ m(t_a)$. We know from the definition of φ_{AS} that either...

- $\mathcal{M}(P^A, \mathcal{S}up(P^S, c), m) = t_r\ m(t_a)$ in which case since $\mathcal{S}up(P^S, c) = \mathcal{S}up(P^A, c)$ in φ_{AS} we also know $\mathcal{M}(P^A, \mathcal{S}up(P^A, c), m) = t_r\ m(t_a)$ and from $P^A \vdash c$ we know that $\exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a.\mathcal{M}(P^A, c, m) = t'_r\ m(t'_a)$, thus from the definition of φ_{AS} we know that $\exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a.\mathcal{M}(P^S, c, m) = t'_r\ m(t'_a)$ □
- $\exists ch, \mathcal{S}up(P^S, c) \sqsubseteq c'\ .\ \delta(c', ch) = (m_{call}, m), \mathcal{M}(P^A, c', m_{call}) = \texttt{void}\ m_{call}(t)$ and $t_r\ m(t_a) = \texttt{void}\ m(t)$ in which case $c \sqsubseteq \mathcal{S}up(P^S, c) \sqsubseteq c'$ so by the same rule, $\mathcal{M}(P^S, c, m) = t_r\ m(t_a)$ i.e. $\exists t'_r = t_r, t'_a = t_a.\mathcal{M}(P^S, c, m) = t'_r\ m(t'_a)$ and the rest follows due to the reflexivity of $(\sqsubseteq)$. □

Lemma 5.3.3 *Well-typedness of expressions is preserved over the translation:*
$P^A, \Gamma \vdash \boldsymbol{e}^A : t \Longrightarrow P^S, \Gamma \vdash \boldsymbol{e}^A : t$

Proof: Induction over the structure of derivations.

Only interesting case is (STSub).

Lemma 5.3.4 *If* σ *is a substitution of argument variables,* $P, \Gamma \vdash \boldsymbol{e} : t \Longrightarrow P, \sigma(\Gamma) \vdash \sigma(\boldsymbol{e}) : t$

Proof: Induction over the structure of derivations.

Only interesting case is (STVar) where we case for the argument either being in the substitution or not.

Lemma 5.3.5 $P^A \vdash c \Longrightarrow P^S \vdash c$

Proof:

$$\begin{array}{lll} P^A \vdash c & \Longrightarrow & P^A \vdash \mathcal{S}up(P^A, c) \diamond_{cl} \quad \text{(WFClass)} \\ & \Longrightarrow & P^A \vdash \mathcal{S}up(P^S, c) \diamond_{cl} \quad \text{(Def } \varphi_{AS}\text{)} \\ & \Longrightarrow & P^S \vdash \mathcal{S}up(P^S, c) \diamond_{cl} \quad \text{(Lemma 5.3.1)} \end{array}$$

$$\begin{array}{lll} P^A \vdash c & \Longrightarrow & \forall m. \mathcal{M}(P^S, \mathcal{S}up(P^S, c), m) = msig \neq \mathcal{U}df \Longrightarrow \\ & & \mathcal{M}(P^S, c, m) = msig \quad \text{(Lemma 5.3.2)} \end{array}$$

For all methods m and chords $(\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{SCh}s(P^S, c, m)$ either

- $(\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{SCh}s(P^A, c, m)$ where $\mathbf{e} \in SrcExpr^A$, thus $\mathbf{e} \in SrcExpr^S$, in which case:
  - $\forall i \in \{1 \ldots n\}$. $\mathcal{M}(P^A, c, m_i) = \texttt{void}\ m_i(t_i)$ and since m_i is not "new", $\mathcal{M}(P^S, c, m_i) = \texttt{void}\ m_i(t_i)$ (Def φ_{AS}).
  - $\mathcal{M}(P^A, c, m) = t\ m(t_0)$ and so $\mathcal{M}(P^S, c, m) = t\ m(t_0)$ by the same argument.
  - $P^A, \Gamma \vdash \mathbf{e} : t$ (where $\Gamma = [m_1\texttt{\_x} \mapsto t_1 \ldots m_n\texttt{\_x} \mapsto t_n, m\texttt{\_x} \mapsto t_0, \texttt{this} \mapsto c]$) and so $P^S, \Gamma \vdash \mathbf{e} : t$ (lemma 5.3.3).
- $ch \in \mathcal{ACh}s(P^A, c)$, $\delta(c, ch) = (m_{call}, m)$, $\{m_1 \ldots m_n\} = ch{\downarrow_1} \backslash \{m_{call}\}$, $\mathbf{e} = S(ch_{\downarrow 2})$ where $S = [m\texttt{\_x}/m_{call}\texttt{\_x}]$), in which case:
  - $\forall i \in \{1 \ldots n\}$. $\mathcal{M}(P^A, c, m_i) = \texttt{void}\ m_i(t_i)$ and since m_i is not "new", $\mathcal{M}(P^S, c, m_i) = \texttt{void}\ m_i(t_i)$ (Def φ_{AS}).
  - Since $\delta(c, ch) = (m_{call}, m)$, $\mathcal{M}(P^S, c, m) = \texttt{void}\ m(t_{call})$ where t_{call} such that $\mathcal{M}(P^A, c, m_{call}) = \texttt{void}\ m(t_{call})$ (Def φ_{AS}). We know $m_{call} \in ch{\downarrow_1}$ (Def δ).
  - $P^A, \Gamma \vdash \mathbf{e} : t$ (where $\Gamma = [m_1\texttt{\_x} \mapsto t_1 \ldots m_n\texttt{\_x} \mapsto t_n, m_{call}\texttt{\_x} \mapsto t_{call}, \texttt{this} \mapsto c]$) and so $P^S, S(\Gamma) \vdash S(ch_{\downarrow 2}) : t$ (lemma 5.3.4) where $S = [m\texttt{\_x}/m_{call}\texttt{\_x}]$.
- $ch \in \mathcal{ACh}s(P^A, c)$, $\delta(c, ch) = (m, m')$, $\{m_1 \ldots m_n\} = \emptyset$, $\mathbf{e} = \texttt{spawn this}.m'(m\texttt{\_x})$ in which case:
  - Let t, t_0 such that $\mathcal{M}(P^A, c, m) = t\ m(t_0)$. $\mathcal{M}(P^S, c, m) = t\ m(t_0)$ as seen before.
  - We can show that $P^S, [m\texttt{\_x} \mapsto t_0, \texttt{this} \mapsto c] \vdash \texttt{spawn this}.m'(m\texttt{\_x}) : t$ because $m \in ch{\downarrow_1}$ (Def δ) ensures m returns $t = \texttt{void}$ and $\mathcal{M}(P^S, c, m') = \texttt{void}\ m'(t_0)$ (Def φ_{AS}).

Since $\mathcal{ACh}s(P^S, c) = \emptyset$, the last part of (WFClass) is always true.
The above is sufficient to prove $P^S \vdash c$. □

Theorem 5.3.6 $\vdash P^A \implies \vdash P^S$.

Proof:

$$
\begin{array}{rcll}
\vdash P^A & \implies & \forall c. \mathcal{S}up(P^A, c) \neq \mathcal{U}df \Rightarrow P^A \vdash c & \text{(WFProg)} \\
& \implies & \forall c. \mathcal{S}up(P^S, c) \neq \mathcal{U}df \Rightarrow P^A \vdash c & \text{(Def } \varphi_{AS}\text{)} \\
& \implies & \forall c. \mathcal{S}up(P^S, c) \neq \mathcal{U}df \Rightarrow P^S \vdash c & \text{(Lemma 5.3.5)} \\
& \implies & \vdash P^S & \text{(WFProg)}
\end{array}
$$

5.4 Preservation of behaviour (proof)

This section documents an incomplete attempt to prove that the behaviour of a program and its translated program are the same. This section is incomplete because we have not fully identified the property that needs to be proved, and the incomplete proof in the final part is not a strong enough property. In order to complete this section, we need to review exactly what property correctly specifies a behavioural equivalence relation, and this will need more research into related work that there was time to complete during this project.

This section has been left in the report because although incomplete, it gives insight into the behaviour of translated programs, when compared to the original programs. This insight was gained through observing the execution of the translation of an example program, and how we can show this translation is bisimilar to the original program.

5.4.1 Motivation

Proving that the behaviour of the translated code is equivalent to the original was much harder than verifying any of the other properties. Firstly, there are many definitions of what constitutes equivalent behaviour. We wanted to show a kind of observational equivalence, where the result and effect of executing the translated program (i.e. the effect on the heap) could be shown to be equivalent in some *reasonable* way to the result of the original program.

For instance we could consider programs to be equivalent if they produce the same results, i.e. we would consider the actual path of execution to be irrelevant. However we might consider the effect of scaling the input of the program to be observable, and thus we would want to distinguish programs by their efficiency, or complexity (e.g. most people consider bubblesort to be different to quicksort).

At the other end of the scale we could consider programs to be equivalent only if every aspect of their execution was identical. In summary, we must justify our decision about what subset of behaviours we consider are "observed", and show that the translation preserves these behaviours.

In particular, we wanted to show that the concurrency of the program was preserved, i.e. the non-determinacy (forking) due to the possible interleavings of execution steps in each thread, the unordering in the queue, and the selection of which chord to invoke when more than one is possible, was not serialised in the translation.

Preserving concurrency is important, because the program in question may be designed so it can be executed in a multi-threaded environment where the threads will run in parallel. The code is designed to execute efficiently in this environment and we must show that the translation preserves the programmer's design. We consider preserving parallelism to be

as important as complexity, and we would expect a translation of quicksort to preserve the $O(n \log(n))$ complexity.

It is important that the translated program should not exhibit *more* behaviours than the original program. For instance we may be introducing race conditions. To illustrate – if the original program is capable of driving someone to work every day, we would not want the translation to be capable of the same behaviour, but also the (very observable) behaviour of driving into a wall. We must show that the observable behaviour of the translation is both sound and complete with respect to the original program.

5.4.2 Example

In order to demonstrate that the translation does indeed preserve the non-determinism of the original program, and does not add any new behaviours as well, we show how the execution of a simple program and its translation proceeds.

This example program has been chosen because it has a non-deterministic behaviour. We are particularly interested in the non-determinism due to the choice of executing a method in a synchronous or asynchronous way, since this is the only part of the behaviour of a program which is affected by the translation of that program (the translation turns asynchronous methods into synchronous ones). It is instructive to see how the behaviour of the translation compares to the original.

Original program P^A

```
async m2(t2 m2_x) {
        e_a;
}
void m1(t1 m1_x) & async m2(t2 m2_x) {
        e_s;
}
```

Translation P^S

```
void m2(t2 m2_x) {
        spawn this.m2'(m2_x);
}
void m2'(t2 m2'_x) {
        e_a;
}
void m1(t1 m1_x) & async m2(t2 m2_x){
        e_s;
}
```

$\delta(c, (\{m_2\}, e_a)) = (m_2, m')$

We can now observe the execution of the single runtime expression $\iota.m_2(v_2); \iota.m_1(v_1)$ in a heap h where $h = [\iota \mapsto [\![c \| [m_2 \mapsto \emptyset]]\!]]$. For clarity, we omit the ι from the runtime expressions, and represent the heap $[\iota \mapsto [\![c \| [m_2 \mapsto V]]\!]]$ with $[\![V]\!]$. The possible executions are shown in figure 5.2. We can see that the net result of executing both programs (the leaves of the trees) is the same.

5.4.3 Early decision-making

One interesting property of the translation in general, that is illustrated by this example, is that the choice as to whether to invoke the synchronous or asynchronous chord with the call to $m_2(v_2)$ is made "one step earlier" in the translated version. This is because while in $\mathcal{L}^A$ there is choice of two "uses" of the value v_2 in the queue, in $\mathcal{L}^S$ there is a choice as to whether to invoke the call to m_2 asynchronously, and put the value in the queue, or whether to call they synchronous chord of m_2 with the value (which does not involve the queue at all).

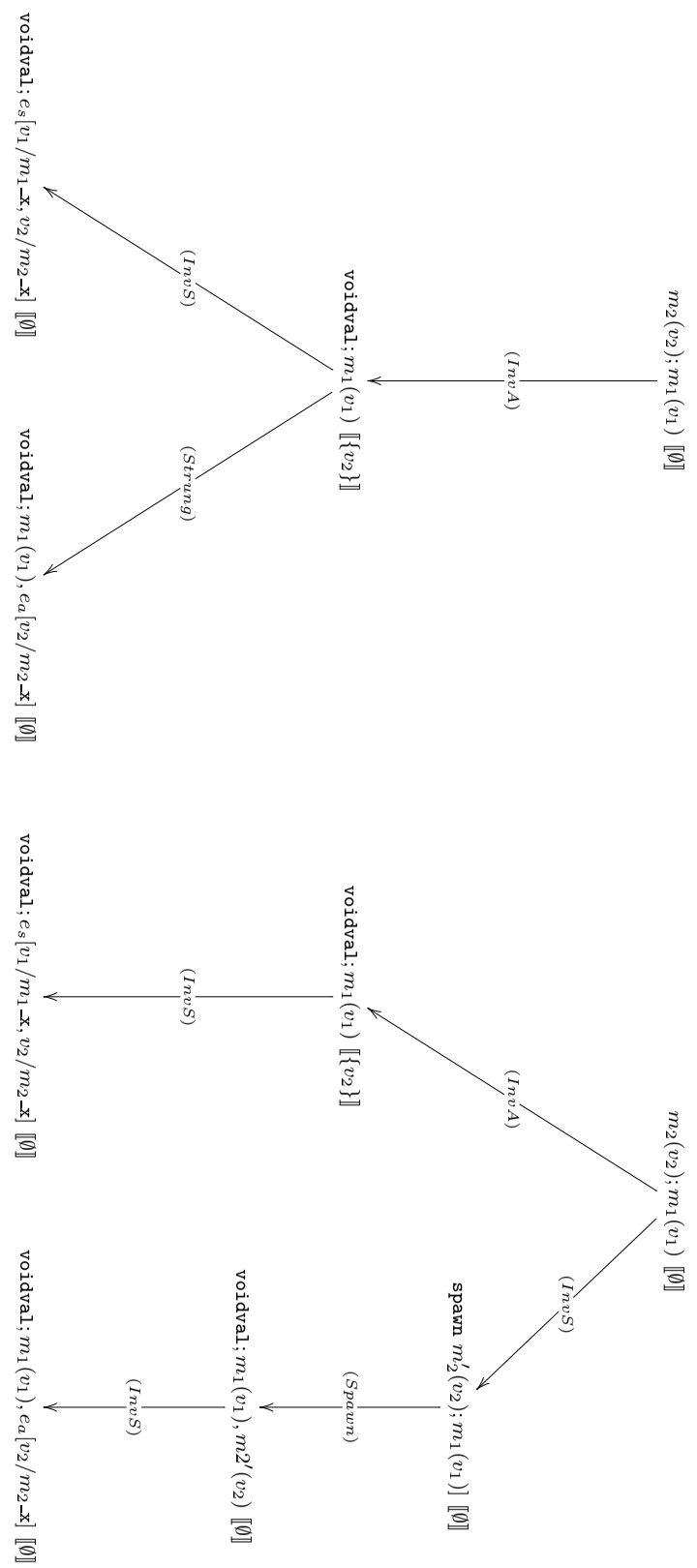


Figure 5.2: Execution of small example program.

5.4.4 Bisimilarity

Bisimilarity is a method for comparing the concurrent behaviour of two programs. We say two programs are bisimilar if there is an equivalence relation $(\sim) \subseteq S^A \times S^S$ between the states of each program (we abstract from the actual definition of the runtime state for $\mathcal{L}^A$ and $\mathcal{L}^S$ with $S^A = Multiset(RExpr^A) \times Heap$ and $S^S = Multiset(RExpr^S) \times Heap$) that satisfies the following properties:

$$\forall s_1^A, s_2^A, s_1^S \;.\; s_1^A \rightsquigarrow s_2^A \;\wedge\; s_1^A \sim s_1^S \implies \exists s_2^S \;.\; s_1^S \rightsquigarrow s_2^S \;\wedge\; s_2^A \sim s_2^S$$
$$\forall s_1^S, s_2^S, s_1^A \;.\; s_1^S \rightsquigarrow s_2^S \;\wedge\; s_1^A \sim s_1^S \implies \exists s_2^A \;.\; s_1^A \rightsquigarrow s_2^A \;\wedge\; s_2^A \sim s_2^S$$

For the example given in figure 5.2, we can define an appropriate equivalence relation as shown in figure 5.3. It works by relating the nodes where decisions occur. This works in the above example, but it has a strange property: We know that $S^A \subset S^S$, so it is possible that $\exists s^A \in S^S$. With this equivalence, $s^A \nsim s^A$, for example $\texttt{voidval}; m_1(v_1)\ [\![\{v_2\}]\!] \nsim \texttt{voidval}; m_1(v_1)\ [\![\{v_2\}]\!]$.

Unfortunately we could not find a way to generalise this bisimulation to show that every program and its translation are bisimilar. We have however proved a similar property with a much simpler equivalence relation.

5.4.5 A Simpler equivalence relation

This is illustrated in figure 5.4. Firstly note that for all s^A, $s^A \sim s^A$. The idea is to relate states if the information contained in their expressions and the heap are equivalent.

In the translated program, when an asynchronous method is called, values are sometimes not placed on the queue, the call can instead be synchronous and the value gets embedded in the body of the invoked synchronous chord. To account for this, we define "pseudo-queues". Pseudo-queues contain all the values that were not placed on the queues during the execution of the translated program. Pseudo-queues exist in two forms:

- As extra threads in the state of the translated program. These threads are just deterministic invocations of the synchronous chord that wraps the around the body of an asynchronous chord in the original program. The missing value is contained as the argument to this method call.
- As `spawn` statements that when executed, will create a thread as described above. Aside from the `spawn` statement, the thread has a corresponding thread in the original program's state.

These two conditions account for the extra two states in the example of figure 5.4 that were equivalent to $\texttt{voidval}; m_1(v_1)\ [\![\{v_2\}]\!]$. To capture the flavour of this equivalence, for each state s^A where there are n queues and m_i ($i \in \{1 \ldots n\}$) distinct values in each, there will be $\Sigma_{i \in \{1 \ldots n\}} 3^{m_i}$ equivalent states in S^S, as each value can be accounted for in 3 ways.

The equivalence relation is formalised in figure 5.5. Unfortunately this equivalence is not a bisimilarity since not all equivalent states have the same number of outcomes (because the decision as to which "outcome" is made earlier in the execution). We can show the bisimulation property from the translation to the original, i.e.

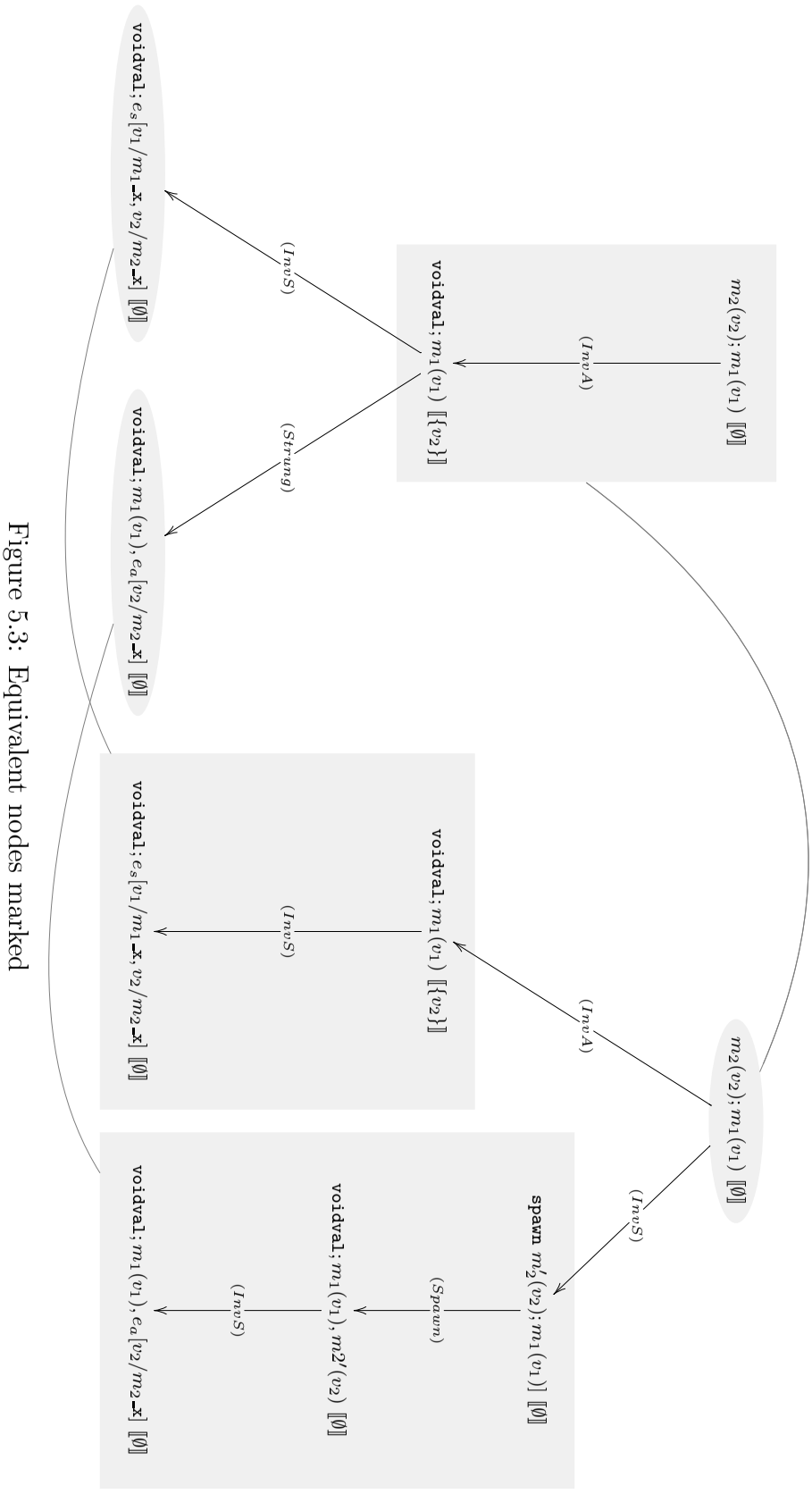


Figure 5.3: Equivalent nodes marked

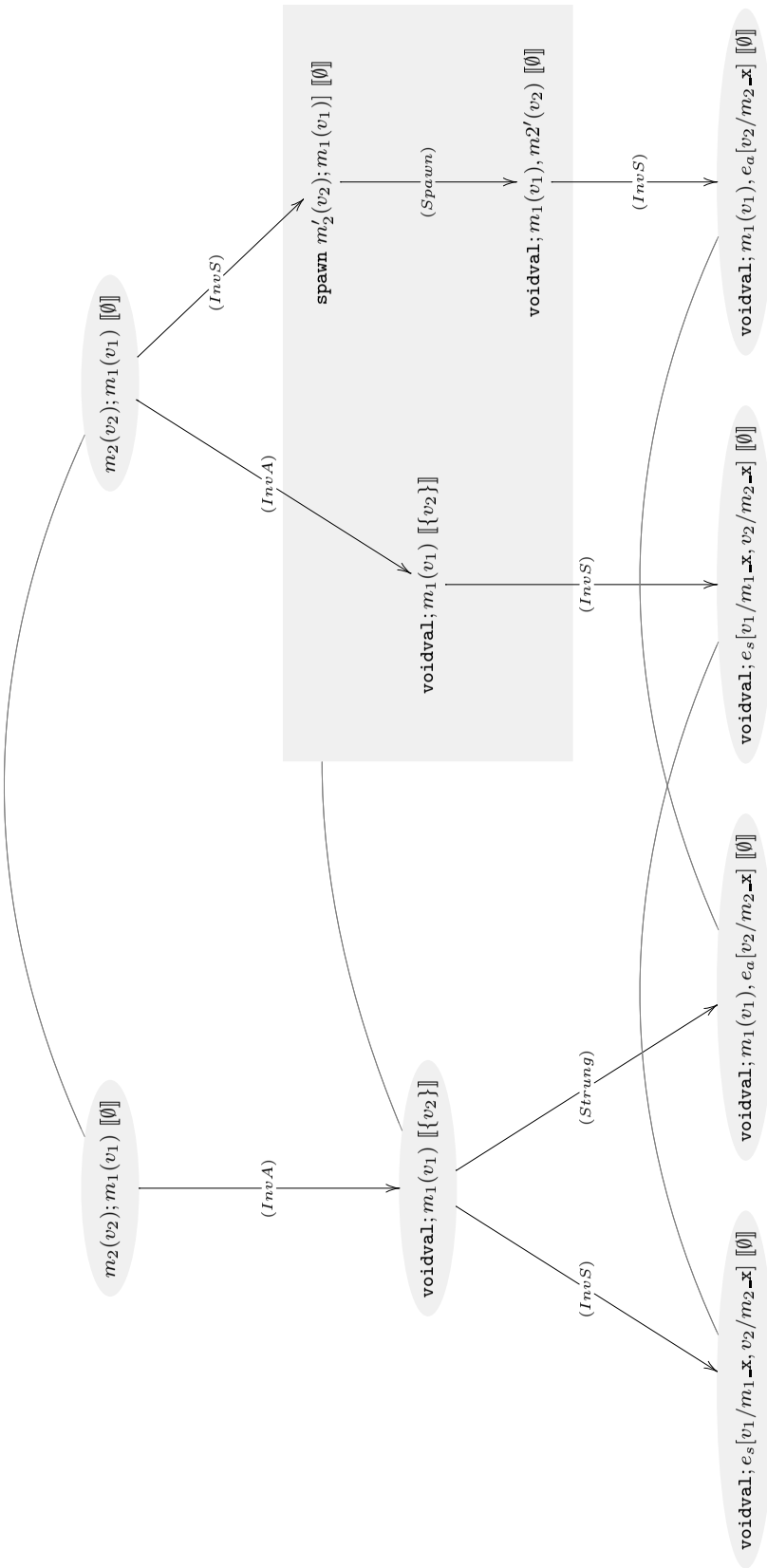


Figure 5.4: Equivalent nodes marked

$$\frac{\begin{array}{l}\forall \iota \in dom(h^A) \cup dom(h^S)\ . \\ \quad \text{there are } c, qs^A, qs^S \text{ such that } h^A(\iota) = [\![c \| qs^A]\!],\ h^S(\iota) = [\![c \| qs^S]\!] \text{ that satisfy} \\ \quad \forall m \in dom(qs^A) \cup dom(qs^S)\ . \\ \qquad \delta, c, m, \iota \vdash T^A, qs^A(m) \sim T^S, qs^S(m)\ \diamond_Q\end{array}}{\delta \vdash T^A, h^A \ \sim\ T^S, h^S} \quad \text{(EqState)}$$

$$\text{there exists } PQ_1 \text{ such that } \forall i \in \{1 \dots n\}\ .$$
$$PQ_1(i) = \begin{cases} \{v\} & \text{if } e_i^A = E[\texttt{voidval}], e_i^S = E[\texttt{spawn}\ \iota.m'(v)], \\ & \qquad \exists ch\ .\ \delta(c, ch) = (m, m') \\ \emptyset & \text{if } e_i^A = E[\texttt{voidval}], e_i^S = E[\texttt{spawn}\ \iota.m'(v)], \\ & \qquad \exists ch\ .\ \delta(c, ch) = (m_{call}, m'), m_{call} \neq m \\ \emptyset & \text{if } e_i^A = e_i^S \end{cases} \quad \text{(EqQueue)}$$

$$\text{there exists } PQ_2 \text{ such that } \forall i \in \{n+1 \dots n+k\}\ .$$
$$PQ_2(i) = \begin{cases} \{v\} & \text{if } e_i^S = \iota.m'(v), \exists ch\ .\ \delta(c, ch) = (m, m') \\ \emptyset & \text{if } e_i^S = \iota.m'(v), \exists ch\ .\ \delta(c, ch) = (m_{call}, m'), m_{call} \neq m \end{cases}$$

$$\frac{V^A = V^S \ \uplus\ \biguplus_i PQ_1(i) \ \uplus\ \biguplus_i PQ_2(i)}{\delta, c, m, \iota \vdash e_1^A \dots e_n^A, V^A \ \sim\ e_1^S \dots e_n^S, e_{n+1}^S \dots e_{n+k}^S, V^S\ \diamond_Q}$$

Figure 5.5: Simple equivalence relation where $\uplus$ is multiset union. T denotes a multiset of threads. V denotes a multiset of values, i.e. a single queue.

$$\forall s_1^S, s_2^S, s_1^A\ .\ P^S \vdash s_1^S \rightsquigarrow s_2^S\ \wedge\ \delta \vdash s_1^A \sim s_1^S\ \implies\ \exists s_2^A\ .\ P^A \vdash s_1^A \rightsquigarrow s_2^A\ \wedge\ \delta \vdash s_2^A \sim s_2^S$$

However in the other direction we can only show:

$$\forall s_1^A, s_2^A\ .\ P^A \vdash s_1^A \rightsquigarrow s_2^A\ \implies\ \exists s_1^S, s_2^S\ .\ \delta \vdash s_1^A \sim s_1^S\ \wedge\ P^S \vdash s_1^S \rightsquigarrow s_2^S\ \wedge\ \delta \vdash s_2^A \sim s_2^S$$

The difference between the two properties is more clearly expressed in this illustrative notation: The reason we can prove $\forall$ from the translation to the original program is that there is only *one* equivalent state in the original program, for each state in the translation.

$$\forall \rightsquigarrow^S\ .\ \forall \sim^1\ .\ \exists \rightsquigarrow^A\ .\ \sim^2 \qquad \text{(bisimulation)}$$
$$\forall \rightsquigarrow^A\ .\ \exists \sim^1\ .\ \exists \rightsquigarrow^S\ .\ \sim^2 \qquad \text{(our property)}$$

We feel that this our property is as useful as bisimulation, since the set of states in the translated program that are equivalent to a state in the original program cannot be distinguished from each other by the user or the programmer.

5.4.6 Proof of forwards equivalence

Where $P^S = \varphi_{AS}(P^A, \delta)$ we prove by induction over $P^A \vdash (\rightsquigarrow)$:

$$\forall s_1^A, s_2^A\ .\ P^A \vdash s_1^A \rightsquigarrow s_2^A\ \implies\ \exists s_1^S, s_2^S\ .\ \delta \vdash s_1^A \sim s_1^S\ \wedge\ P^S \vdash s_1^S \rightsquigarrow s_2^S\ \wedge\ \delta \vdash s_2^A \sim s_2^S$$

Case of (Strung)

$s_1^A = T^A, h^A$

$s_2^A = T^A \uplus \{e'^A\}, h'^A$ where
$e'^A = \mathbf{e}^A[v_1/m_1\_\mathtt{x} \ldots v_n/m_n\_\mathtt{x}, \iota/\mathtt{this}]$
$h^A(\iota) = [\![c \| qs]\!]$
$(\{m_1 \ldots m_n\}, \mathbf{e}^A) \in \mathcal{ACHS}(P^A, c)$ (the chord that was strung)
$\forall i \in \{1 \ldots n\}.qs^A(m_i) = \{v_i\} \uplus q_i^A$
$h'^A = h^A[\iota \mapsto [\![c \| qs^A[m_1 \mapsto q_1^A \ldots m_n \mapsto q_n^A]]\!]]$

Since $\{m_1 \ldots m_n\}$ is unordered, we let $\delta(c, (\{m_1 \ldots m_n\}, \mathbf{e}^A)) = (m_1, m')$.

Let $s_1^S = T^S, h^S$. We may choose any s_1^S so long as $\delta \vdash s_1^A \sim s_1^S$.
We know that $\forall \iota' \in dom(h^A) \cup dom(h^S)$.
$h^A(\iota') = [\![c' \| qs^A]\!]$, $h^S(\iota') = [\![c' \| qs^S]\!]$
$\forall m \in dom(qs^A) \cup dom(qs^S)$.
$\delta, c', m, \iota \vdash T^A, qs^A(m) \sim T^S, qs^S(m)\ \diamond_Q$

We leave open how $\delta, c', m, \iota' \vdash T^A, qs^A(m) \sim T^S, qs^S(m)\ \diamond_Q$ for all $\iota' \neq \iota$, or $m \notin \{m_1 \ldots m_n\}$.

For $\delta, c, m_1, \iota \vdash T^A, qs^A(m_1) \sim T^S, qs^S(m_1)\ \diamond_Q$ we recall that $qs^A(m_1) = \{v_1\} \uplus q_1^A$

Let $V^A = qs^A(m_1)$ and $V^S = qs^S(m_1)$. We choose any PQ_1 and PQ_2 so long as $v_1 \in \biguplus_i PQ_2(i)$, i.e. $\exists i.v_1 \in PQ_2(i)$. Informally this means that this value is not stored in a queue in the translated program's state (i.e. not in V^S) but in a "pseudo-queue". Formally, it exists within an expression $\iota.m'(v_1) \in T^S$.

For all $i \in \{2 \ldots n\}$, $\delta, c, m_i, \iota \vdash T^A, qs^A(m_i) \sim T^S, qs^S(m_i)\ \diamond_Q$ we recall that $qs^A(m_i) = \{v_i\} \uplus q_i^A$

Let $V^A = qs^A(m_i)$ and $V^S = qs^S(m_i)$. We choose any PQ_1 and PQ_2 so long as $v_i \in V^S$. Informally this means that this value is stored in a "real" queue in the translated program's state, just as it is in the original program's state. Thus $qs^S(m_i) = \{v_i\} \uplus q_i^S$.

Now let $T^S = \{e_1 \ldots e_k, \iota.m'(v_1)\}$. We know that $h^S(\iota) = [\![c \| qs^S]\!]$. We know from the definition of φ_{AS} that since $(\{m_1 \ldots m_n\}, \mathbf{e}^A) \in \mathcal{ACHS}(P^A, c)$ (from earlier) and $\delta(c, (\{m_1 \ldots m_n\}, \mathbf{e}^A)) = (m_1, m')$ (from earlier) that $(\{m_2 \ldots m_n\}, \mathbf{e}^A[m'\_\mathtt{x}/m_1\_\mathtt{x}]) \in \mathcal{SCHS}(P^S, c, m')$.

We also know that $\forall i \in \{2 \ldots n\}.qs^S(m_i) = \{v_i\} \uplus q_i^S$ and if we let $h'^S = h^S[\iota \mapsto [\![c \| qs^S[m_2 \mapsto q_2^S \ldots m_n \mapsto q_n^S]]\!]]$ then by (Run) with an empty context E, and (InvS):

$P^S \vdash e_1^S \ldots e_k^S, \iota.m'(v_1), h^S \rightsquigarrow$
$e_1^S \ldots e_k^S, \mathbf{e}^A[m'\_\mathtt{x}/m_1\_\mathtt{x}][v_2/m_2\_\mathtt{x} \ldots v_n/m_n\_\mathtt{x}, v_1/m'\_\mathtt{x}, \iota/\mathtt{this}], h'^S$

Therefore we choose $s_2^S = T^S \backslash \{\iota.m'(v_1)\} \uplus \{e'^A\}, h'^S$ (e'^A defined earlier).

It remains to show that $\delta \vdash s_2^A \sim s_2^S$, i.e. knowing that $\delta \vdash T^A, h^A \sim T^S, h^S$, showing that:
$\delta \vdash$
$T^A \uplus \{e'^A\}, h^A[\iota \mapsto [\![c \| qs^A[m_1 \mapsto q_1^A \ldots m_n \mapsto q_n^A]]\!]]$
$\sim$
$T^S \backslash \{\iota.m'(v_1)\} \uplus \{e'^A\}, h^S[\iota \mapsto [\![c \| qs^S[m_2 \mapsto q_2^S \ldots m_n \mapsto q_n^S]]\!]]$

Note that the domain of h'^A and h'^S is the same as before.
We must show that $\forall \iota' \in dom(h'^A) \cup dom(h'^S)$.
$h'^A(\iota') = [\![c' \| qs'^A]\!]$, $h'^S(\iota') = [\![c' \| qs'^S]\!]$
$\forall m \in dom(qs'^A) \cup dom(qs'^S)$.
$\delta, c', m, \iota \vdash T^A, qs'^A(m) \sim T^S, qs'^S(m) \diamond_Q$

We know that $dom(qs'^A) \cup dom(qs'^S) = dom(qs^A) \cup dom(qs^S)$. We know from earlier that in some manner, $\delta, c', m, \iota' \vdash T^A, qs^A(m) \sim T^S, qs^S(m) \diamond_Q$ for all $\iota' \neq \iota$, or $m \notin \{m_1 \ldots m_n\}$. Since $qs^A(m) = qs'^A(m)$ for these values of m, and the same with qs^S and qs'^S, this transfers to the second equivalence if we let the additional thread e'^A on each side be accounted for with an additional $PQ_1(i) = \emptyset$ and the missing thread on the right hand side is simply unmapping a certain $PQ_2(i) = \emptyset$ which has no effect on the union, and thus the equality of queues.

Where $\iota = \iota'$, $m = m_1$, we have to show that
$\delta, c, m_1, \iota \vdash T^A \uplus \{e'^A\}, qs'^A(m_1) \sim T^S \uplus \{e'^A\} \backslash \{\iota.m'(v_1)\}, qs'^S(m_1) \diamond_Q$

We know that $qs^A(m_1) = qs'^A(m_1) \uplus \{v_1\}$ and $qs'^S(m_1) = qs^S(m_1)$. If we take PQ_1 and PQ_2 from the previous equivalence, we need only modify PQ_2 so it no longer accounts for the $\iota.m'(v_1)$, and add an additional $PQ_1(i) = \emptyset$ for the e'^A, since all the other threads are the same. This gives up one less value on the right hand side of the $=$, but note that the same value has also been removed from V^A, so the equivalence still holds.

Where $\iota = \iota'$, $m = m_i$, $i \in \{2 \ldots n\}$, we have to show that
$\delta, c, m_1, \iota \vdash T^A \uplus \{e'^A\}, qs'^A(m_1) \sim T^S \uplus \{e'^A\} \backslash \{\iota.m'(v_1)\}, qs'^S(m_1) \diamond_Q$

We use the same PQ_1 with an extra $PQ_1(k) = \emptyset$ for the extra e'^A on each side. We use the same PQ_2 with a certain $PQ_2(k) = \emptyset$ that used to account for $\iota.m'(v_1)$ removed, neither of which affects the union. We know that $qs'^A(m_i) = qs^A(m_i) \backslash \{v_i\}$ and $qs'^S(m_i) = qs^S(m_i) \backslash \{v_i\}$ so the equality still holds. □

Chapter 6

$\mathcal{L}^S$ is at most as expressive as $\mathcal{L}^A$

We define a translation φ_{SA} from $\mathcal{L}^S$ to $\mathcal{L}^A$. We show that the translation preserves program structure, validity, and behaviour.

6.1 Definition of the translation φ_{SA}

6.1.1 Example

Each `spawn` statement is changed into an invocation of a new asynchronous chord. The expression within the `spawn` statement becomes the body of the chord. Since the original expression could reference *arguments* of the form m_x, `null`, `this`, and `voidval`, we need to ensure these references have the same meaning in the body of the new chord.

The new chord is in the same class as the original `spawn` statement, so the meaning of `this` is preserved. `null` and `voidval` have the same meaning across all methods and classes, but we still have to take care of the method arguments.

To do this, we create our asynchronous chord with one method for each argument that was available to the original `spawn` statement, i.e. one method for each of the methods of the synchronous chord that the `spawn` statement was inside. One of those methods will be synchronous, but all of the new methods will be asynchronous, and thus have a return type of `void`.

At first, we tried replacing `spawn` statements with the sequential composition of all the asynchronous method calls that comprise the new chord. This does not work, however, since if two threads concurrently execute the same ex-spawn site, the two chord invocations arbitrarily swap arguments, causing a large set of new behaviours. We have to use a simple mutex (implemented with chords) at each spawn site.

```
void m1(t1 m1_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
        spawn e;
}
```

We translate to: (The semantics requires an argument for each method in the mutex implementation, but we use a value of `voidval` to represent the fact that it is not used.)

```
init() { this.unlock(voidval) }
```

```
void lock(void lock_x) & async unlock(void unlock_x) { voidval }

void m1(t1 m1_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
        this.lock(voidval);
        this.m1'(m1_x) ; this.m2'(m2_x) ; ... ; this.mn'(mn_x);
        this.unlock(voidval);
}

async m1'(t1 m1'_x) & async m2'(t2 m2'_x) & ... & async mn'(tn mn'_x) {
        e[m1'_x/m1_x ... mn'_x/mn_x];
}
```

Note that the expression `e` and the context of the spawn site (if there was one) are not changed, and neither is the structure of classes or the set of existing methods, which is what we require for macro expressiveness.

6.1.2 Translation

The encoding is given in figure 6.1. Because the encoding can translate into a range of equivalent programs, depending on what we call the new methods, we collect this flexibility into a single "decision" function and specify this as an argument to φ_{AS}.

$$\delta : \Delta = (L \to (Id_c \times Id_m \times Id_m \times \mathbb{P}(Id_m \times Id_m) \times SrcExpr^S)) \times Id_m$$

We also assume that each `spawn` sub-syntax in P^S is labelled with a unique label l, and the set of labels is L. We omit the details of how this labelling is performed, since it is a common technique when processing programs. For each labelled $[\texttt{spawn}\ e]^l$ statement, $\delta{\downarrow_1}(l)$ tells us:

- The class that the `spawn` *s*tatement is defined within.
- The lock() method and
- the unlock() method acting as a mutex for that `spawn` site.
- The mapping of method names to method names, that tells us for each spawn site, what new asynchronous methods "shadow" the methods from the spawn site's original location.
- The body of the `spawn` statement (which must be recursively translated).

By abuse of notation we let $\delta(l) = \delta{\downarrow_1}(l)$. The value of $\delta{\downarrow_2}$ is the name of the method used for initialising the state of the mutex chords in that class.

The result of $\varphi_{SA}(P^S, \delta)$ is only defined for when δ is well-defined, i.e. $P^S \vdash \delta$ as defined below:

$\varphi_{SA}(P^S, \delta) = P^A$ if and only if

$$
\begin{array}{lcl}
\mathcal{SC}hs(P^A, c, m) & = & \{\ (ch{\downarrow_1}, \mathbb{C}(ch{\downarrow_2})) \mid ch \in \mathcal{SC}hs(P^S, c, m)\ \} \\
& \cup & \{\ (unlock, \texttt{voidval}) \mid \exists l.\delta(l) = (c, \_, m, unlock, \_, \_)\ \} \\
& \cup & \{\ (\emptyset, \texttt{this}.unlock_1(\texttt{voidval}); \ldots; \texttt{this}.unlock_n(\texttt{voidval}); \texttt{this}) \\
& & \mid \text{if } m = \delta{\downarrow_2}, \text{ and } \{unlock_1 \ldots unlock_n\} = \{\delta(l){\downarrow_3} | \delta(l){\downarrow_1} = c\}\ \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{AC}hs(P^A, c) & = & \{\ (\{m'_1 \ldots m'_n\}), \texttt{this}.unlock(\texttt{voidval}); \mathbb{C}(e)[m'_1\texttt{\_x}/m_1\texttt{\_x} \ldots m'_n\texttt{\_x}/m_n\texttt{\_x}]) \\
& & \mid l \in L, \delta(l) = (c, \_, \_, unlock, \{(m_1, m'_1) \ldots (m_n, m'_n)\}, e)\ \}
\end{array}
$$

$$\mathcal{S}up(P^A, c) = \mathcal{S}up(P^S, c)$$

$$
\mathcal{M}(P^A, c, m) = \begin{cases}
\mathcal{M}(P^S, c, m) & \text{if } \mathcal{M}(P^S, c, m) \neq \mathcal{U}df \\
\texttt{void}\ m(t) & \text{if } \exists l.c \sqsubseteq \delta(l){\downarrow_1}, (m_{sh}, m) \in \delta(l){\downarrow_4}, \\
& \quad \mathcal{M}(P^S, \delta(l){\downarrow_1}, m_{sh}) = \_\ m_{sh}(t) \\
\texttt{void}\ m(\texttt{void}) & \text{if } \exists l.c \sqsubseteq \delta(l){\downarrow_1}, m \in \{\delta(l){\downarrow_2}, \delta(l){\downarrow_3}\} \\
c\ m(\texttt{void}) & \text{if } m = \delta{\downarrow_2}
\end{cases}
$$

$$
\begin{array}{lcl}
\mathbb{C}(\texttt{null}) & = & \texttt{null} \\
\mathbb{C}(\texttt{this}) & = & \texttt{this} \\
\mathbb{C}(\texttt{voidval}) & = & \texttt{voidval} \\
\mathbb{C}(m\texttt{\_x}) & = & m\texttt{\_x} \\
\mathbb{C}(\texttt{new}\ c) & = & \texttt{new}\ c.\delta{\downarrow_2}(\texttt{voidval}) \\
\mathbb{C}(e_1; e_2) & = & \mathbb{C}(e_1); \mathbb{C}(e_2) \\
\mathbb{C}(e_1.m(e_2)) & = & \mathbb{C}(e_1).m(\mathbb{C}(e_2)) \\
\mathbb{C}([\texttt{spawn}\ e]^l) & = & \texttt{this}.lock(\texttt{voidval}); \texttt{this}.m'_1(m_1\texttt{\_x}); \ldots; \texttt{this}.m'_n(m_n\texttt{\_x}) \\
& & \text{where } \delta(l) = (c, lock, \_, \{(m_1, m'_1) \ldots (m_n, m'_n)\}, \_)
\end{array}
$$

Figure 6.1: Translation $\varphi_{SA} : (\mathcal{L}^S \times \Delta) \to \mathcal{L}^A$

$$
\begin{array}{lcl}
P^S \vdash \delta & \iff & \forall(\{m_1, \ldots m_n\}, e_{ch}) \in \mathcal{SC}hs(P^S, c, m), [\texttt{spawn}\ e]^l \in e_{ch}\ . \\
& & \quad \text{let } \delta(l) = (c, lock, unlock, \{(m_1, m'_1) \ldots (m_n, m'_n), (m, m')\}, e) \\
& & \quad init = \delta{\downarrow_2} \\
& & \quad |\{init, lock, unlock, m'_1 \ldots m'_n, m'\}| = n + 4 \qquad \text{(all are distinct)} \\
& & \quad \forall m'' \in \{init, lock, unlock, m'_1 \ldots m'_n, m'\}\ . \\
& & \qquad \forall c'.\mathcal{M}(P^S, c', m'') = \mathcal{U}df \\
& & \qquad \forall l'\ .\ m'' \in \{\delta(l'){\downarrow_2}, \delta(l'){\downarrow_3}\} \cup \{m''' | (\_, m''') \in \delta(l'){\downarrow_4}\} \implies l = l' \\
& & \forall l, c, m_1 \ldots m_n, m, e\ .\ \delta(l) = (c, \_, \_, \{(m_1, \_) \ldots (m_n, \_), (m, \_)\}, e) \implies \\
& & \qquad \exists e_{ch} \ni [\texttt{spawn}\ e]^l.(\{m_1, \ldots m_n\}, e_{ch}) \in \mathcal{SC}hs(P^S, c, m)
\end{array}
$$

This enforces the idea that the new method names are distinct from existing methods, and each other, across the entire program (distinction within a class is not sufficient since method names can still interfere due to the inheritance of methods).

It also ensures the set of method pairs is such that there is one pair for each method in the enclosing chord, and the first element of the pair is that method. This is used like a look-up table in the translation, so we know which method is "shadowing" each method in the chord.

The value of δ used in the above example translation of a P^S program was $\delta = ([l \mapsto (c, lock, unlock, \{(m_1, m_1') \ldots (m_n, m_n')\}, e)], init)$.

6.1.3 Properties of the translation

These properties are not interesting results in the broader scope of this report, but are presented here because they reflect the characteristics of this translation, and thus help us understand it. If $P^A = \varphi_{SA}(P^S, \delta)$:

- $\forall c \,.\, \mathcal{Q}(P^S, c) \supseteq \mathcal{Q}(P^A, c)$ No queues are lost over the translation, but many are added because each the asynchronous chord that implements the `spawn` statement has a set of new asynchronous methods. Also there is an asynchronous method `unlock()` as part of the mutex that we use to guard each invocation of an asynchronous chord.

- From $P^A \vdash \delta$, we know that the definition of $\mathcal{M}(P^S, c, m)$ is unambiguous, although it is possible that none of the cases apply.

6.2 Preservation of structure (proof)

We first formally define the property of structure preservation for our programs P^A and P^S where $P^A = \varphi_{SA}(P^S, \delta)$. We refer to section 3.1.2, where the contribution of [5] is put into the context of our object-oriented formalisations. In order for the translation to be structure-preserving, it must satisfy:

- The only construct in the source expressions of $\mathcal{L}^S$ that is not present in $\mathcal{L}^A$ is `spawn` **e**. All the other constructs must not be changed in the translation. This means that the method signatures must remain intact as otherwise it would not be possible to prove well-formedness of P^S since method calls must be left as they are. `spawn` **e** statements must be replaced with program code that contains a single copy of **e** but is otherwise independent of **e**.

  We can see from the definition of $\mathbb{C}$ that the only constructs over which $\mathbb{C}$ is not homomorphic are `new` c and `spawn` **e**. `new` c is an exception to this proof that was justified in section 3.1.4. The `spawn` **e** construct is changed to a construct that does not use **e**, but the **e** is used unchanged in the body of an asynchronous chord, excepting the translation of argument variables that is justified in section 3.1.4. □

- Method signatures are common to $\mathcal{L}^A$ and $\mathcal{L}^S$, so they must not be altered, although we do allow addition of new methods.
  $\forall c, m. \mathcal{M}(P^S, c, m) = msig \Rightarrow \mathcal{M}(P^A, c, m) = msig$

  From $P^S \vdash \delta$ we know that the four cases for the definition of $\mathcal{M}(P^S, c, m)$ in φ_{AS} are exclusive, so when $\mathcal{M}(P^A, c, m) = msig$ i.e. $\mathcal{M}(P^A, c, m) \neq \mathcal{U}\mathcal{d}\mathcal{f}$, we know that $\mathcal{M}(P^S, c, m) = \mathcal{M}(P^A, c, m) = msig$. □

- Synchronous chords are common to $\mathcal{L}^S$ and $\mathcal{L}^A$ so the synchronous chords in a class must be preserved, aside from the recursion of the translation into their bodies. We

do, however, allow new synchronous chords can be added:
$\forall c, m.\ (\{m_1 \ldots m_n\}, \mathbf{e}^S) \in \mathcal{SCHS}(P^S, c, m) \Longrightarrow (\{m_1 \ldots m_n\}, \mathbb{C}(\mathbf{e}^S)) \mathcal{SCHS}(P^A, c, m)$

This can be seen from inspection of the definition of $\mathcal{SCHS}(P^A, c, m)$ in the translation. □

- The feature of classes is common to $\mathcal{L}^A$ and $\mathcal{L}^S$, so the structure of classes must be preserved, although we allow the addition of new classes:
$\forall c. \mathcal{Sup}(P^S, c) = c' \Longrightarrow \mathcal{Sup}(P^A, c) = c'$

  We can prove this by inspection of φ_{SA}, which actually has the stronger property that no classes are added or removed. □

6.3 Preservation of well-formedness (proof)

We need to establish that the well-formedness of the translated source code is preserved over translation, this is the notion of programness in [5]. Let $P^A = \varphi_{SA}(P^S, \delta)$. We must show that $\vdash P^S \Longrightarrow \vdash P^A$.

Lemma 6.3.1 *If $P^A = \varphi_{SA}(P^S, \delta)$ then $P^S \vdash c \diamond_{cl} \Longrightarrow P^A \vdash c \diamond_{cl}$*

Proof:

$$\begin{array}{lcll} P^S \vdash c \diamond_{cl} & \Longrightarrow & \mathcal{Sup}(P^S, c) \neq \mathcal{Udf} \vee c = \texttt{Object} & \text{(IsClass)} \\ & \Longrightarrow & \mathcal{Sup}(P^A, c) \neq \mathcal{Udf} \vee c = \texttt{Object} & \text{(Def } \varphi_{SA}\text{)} \\ & \Longrightarrow & P^A \vdash c \diamond_{cl} & \text{(IsClass)} \end{array}$$

Lemma 6.3.2 *Method type signatures are preserved over inheritance in the translated program: If $\vdash P^S$ and $P^A = \varphi_{SA}(P^S, \delta)$ then*
$\forall m. \mathcal{M}(P^A, \mathcal{Sup}(P^A, c), m) = t_r\ m(t_a) \Longrightarrow \exists t'_r, t'_c\ .\ \mathcal{M}(P^A, c, m) = t'_r\ m(t'_a), t'_r \sqsubseteq t_r, t_a \sqsubseteq t'_a$.

Proof:
Let $\vdash P^S$ and $\mathcal{M}(P^A, \mathcal{Sup}(P^A, c), m) = t_r\ m(t_a)$. We know from the definition of φ_{SA} that there are four possibilities:

- $\mathcal{M}(P^S, \mathcal{Sup}(P^A, c), m) = t_r\ m(t_c) \neq \mathcal{Udf}$ and since $\mathcal{Sup}(P^A, c) = \mathcal{Sup}(P^S, c)$ in φ_{SA} we know $\mathcal{M}(P^S, \mathcal{Sup}(P^S, c), m) = t_r\ m(t_c) \neq \mathcal{Udf}$ in which case from $P^S \vdash c$ we know that $\mathcal{M}(P^S, c, m) = t'_r\ m(t'_c), t'_r \sqsubseteq t_r, t_c \sqsubseteq t'_c$, and from the definition of φ_{SA} we know that $\mathcal{M}(P^A, c, m) = t'_r\ m(t'_c)$ □

- $\exists l. \mathcal{Sup}(P^S, c) \sqsubseteq \delta(l)\downarrow_1, (m_{sh}, m) \in \delta(l)\downarrow_4, \mathcal{M}(P^S, \delta(l)\downarrow_1, m_{sh}) = \_\ m_{sh}(t)$ and $t_r = \texttt{void}$, $t_c = t$. Let $\mathcal{M}(P^A, c, m) = t'_r\ m(t'_a)$. Note that $c \sqsubseteq \mathcal{Sup}(P^S, c)$ and so therefore the same case applies for c so $t'_r = t_r$and $t'_c = t_c$ □

- $\exists l. \mathcal{Sup}(P^S, c) \sqsubseteq \delta(l)\downarrow_1, m \in \{\delta(l)\downarrow_2, \delta(l)\downarrow_3\}$ and we use the same argument as above to show that the same type signature $\texttt{void}\ m(\texttt{void})$ is defined for $\mathcal{M}(P^A, c, m)$. □

- $m = \delta\downarrow_2$ in which case $\mathcal{M}(P^A, \mathcal{S}up(P^A, c), m) = \mathcal{S}up(c)\ m(\texttt{void})$ and also $\mathcal{M}(P^A, c, m) = c\ m(\texttt{void})$. Thus $t'_c = t_c$ and since $c \sqsubseteq \mathcal{S}up(P^S, c)$, $t'_r \sqsubseteq t_r$. □

Lemma 6.3.3 *Well-typedness of expressions is preserved over the translation:*
If $P^A = \varphi_{SA}(P^S, \delta)$,
$(m_1 \ldots m_n, \boldsymbol{e}^S_{ch}) \in \mathcal{SC}\mathcal{hs}(P^S, c, m)$,
$\mathcal{M}(P^S, c, m) = \_\ m(t_0)$, $\mathcal{M}(P^S, c, m_i) = \_\ m_i(t_i)$ *(for all* $i \in \{1 \ldots n\}$*)*,
$\Gamma = [m_1\_\boldsymbol{x} \mapsto t_1 \ldots m_n\_\boldsymbol{x} \mapsto t_n, m\_\boldsymbol{x} \mapsto t_0, \boldsymbol{this} \mapsto c]$, *and*
$\boldsymbol{e}^S$ *is a subterm of* $\boldsymbol{e}^S_{ch}$, *then*
$P^S, \Gamma \vdash \boldsymbol{e}^S : t \Longrightarrow P^A, \Gamma \vdash \mathbb{C}(\boldsymbol{e}^S) : t$

Proof: Induction over the structure of derivations.

- STVar, STNull, STVoid: Trivial. □
- STNew: Note that $\mathbb{C}(\texttt{new}\ c) = \texttt{new}\ c.init(\texttt{voidval})$ and that $\mathcal{M}(P^A, c, init) = c\ m(\texttt{void})$, using STNew and STInv we can show that $P^A, \Gamma \vdash \mathbb{C}(\texttt{new}\ c) : c$. □
- STInv: Use $\mathcal{M}(P^S, c, m) = msig \neq \mathcal{U}df \Longrightarrow \mathcal{M}(P^A, c, m) = msig$. □
- STSeq: Trivial. □
- STSub: Use $\mathcal{S}up(P^A, c) = \mathcal{S}up(P^S, c)$ from definition of φ_{SA}. □
- STSpawn: We have to show that $P^A, \Gamma \vdash \mathbb{C}([\texttt{spawn}\ \mathbf{e}^S_b]^l) : \texttt{void}$. Note that from the premises, and $P^S \vdash \delta$ we know that

  $\delta(l) = (c, lock, \_, \{(m_1, m'_1) \ldots (m_n, m'_n), (m, m')\}, \mathbf{e}^S_b)$

  and from the definition of φ_{SA} we note that

  $\mathbb{C}([\texttt{spawn}\ \mathbf{e}^S_b]^l) = \texttt{this}.lock(\texttt{voidval}); \texttt{this}.m'_1(m_1\_\texttt{x}); \ldots; \texttt{this}.m'_n(m_n\_\texttt{x}); \texttt{this}.m'(m\_\texttt{x})$.

  We must use STSeq to show this is typeable to `void`, and to this end it suffices to show that all the method invocations are typeable to `void`. Using STInv with STVar and $\mathcal{M}(P^A, c, lock) = \texttt{void}\ lock(\texttt{void})$ we know the first method satisfies. We know from the premises that $\mathcal{M}(P^S, c, m) = \_\ m(t_0)$, $\mathcal{M}(P^S, c, m_i) = \_\ m_i(t_i)$ and from the definition of φ_{SA} and $\delta(l)$ we therefore know $\mathcal{M}(P^A, c, m') = \texttt{void}\ m'(t_0)$, $\mathcal{M}(P^A, c, m'_i) = \texttt{void}\ m'_i(t_i)$. This means we can use STInv to type each of the method invocations to `void`in the sequential composition, and it follows that the block is also `void`. □

Lemma 6.3.4 *If* $\sigma : SrcExpr \to SrcExpr$ *is a substitution of argument variables, and* $\forall m\_\boldsymbol{x}.\Gamma'(m\_\boldsymbol{x}) = \Gamma(\sigma(m\_\boldsymbol{x}))$, *then* $P, \Gamma \vdash \boldsymbol{e} : t \Longrightarrow P, \Gamma' \vdash \sigma(\boldsymbol{e}) : t$

Proof: Induction over the structure of derivations.

Only interesting case is (STVar) where we case for the argument either being in the substitution or not.

Lemma 6.3.5 $P^S \vdash c \Longrightarrow P^A \vdash c$

Proof:

$$\begin{array}{lll} P^S \vdash c & \Longrightarrow & P^S \vdash \mathcal{S}up(P^S, c) \diamond_{cl} \quad \text{(WFClass)} \\ & \Longrightarrow & P^S \vdash \mathcal{S}up(P^A, c) \diamond_{cl} \quad \text{(Def } \varphi_{AS}\text{)} \\ & \Longrightarrow & P^A \vdash \mathcal{S}up(P^A, c) \diamond_{cl} \quad \text{(Lemma 6.3.1)} \end{array}$$

$$\begin{array}{lll} P^S \vdash c & \Longrightarrow & \forall m.\mathcal{M}(P^A, \mathcal{S}up(P^A, c), m) = t_r\ m(t_a) \\ & & \Rightarrow \exists t'_r, t'_c\ .\ \mathcal{M}(P^A, c, m) = t'_r\ m(t'_a), t'_r \sqsubseteq t_r, t_a \sqsubseteq t'_a \quad \text{(Lemma 6.3.2)} \end{array}$$

For all methods m and synchronous chords $(\{m_1 \ldots m_n\}, \mathbf{e}^A) \in \mathcal{SCh}s(P^A, c, m)$ either

- $(\{m_1 \ldots m_n\}, \mathbf{e}^S) \in \mathcal{SCh}s(P^S, c, m)$ where $\mathbf{e}^A = \mathbb{C}(\mathbf{e}^S)$, in which case:
  - $\forall i \in \{1 \ldots n\}\ .\ \mathcal{M}(P^S, c, m_i) = \texttt{void}\ m_i(t_i)$ (WFClass) and by the definition of φ_{SA}, $\mathcal{M}(P^A, c, m_i) = \texttt{void}\ m_i(t_i)$.
  - $\mathcal{M}(P^S, c, m) = t\ m(t_0)$ (WFClass) and so $\mathcal{M}(P^A, c, m) = t\ m(t_0)$ by the same argument.
  - $P^S, \Gamma \vdash \mathbf{e}^S : t$ (where $\Gamma = [m_1\texttt{\_x} \mapsto t_1 \ldots m_n\texttt{\_x} \mapsto t_n, m\texttt{\_x} \mapsto t_0, \texttt{this} \mapsto c]$) (WFClass) and so $P^A, \Gamma \vdash \mathbf{e}^A : t$ (lemma 6.3.2).
- $n = 1$, $\delta(l) = (c, m, m_1, \_, \_)$, $\mathbf{e}^A = \texttt{voidval}$ in which case:
  - By the definition of φ_{SA}, we know that $\mathcal{M}(P^A, c, m_1) = \texttt{void}\ m_1(\texttt{void})$ and also $\mathcal{M}(P^A, c, m) = \texttt{void}\ m(\texttt{void})$.
  - Using STVoid, $P^A, \Gamma \vdash \mathbf{e}^A : \texttt{void}$ (where $\Gamma = [m_1\texttt{\_x} \mapsto \texttt{void} m\texttt{\_x} \mapsto \texttt{void}, \texttt{this} \mapsto c]$).
- $m = init$, $n = 0$ and $\mathbf{e}^A = \texttt{this}.unlock_1(\texttt{voidval}); \ldots; \texttt{this}.unlock_n(\texttt{voidval}); \texttt{this}$ where $\{unlock_1 \ldots unlock_n\} = \{\delta(l)\downarrow_3 | \exists l.\delta(l)\downarrow_1 = c\}$
  - We know from the definition of φ_{SA} that $\mathcal{M}(P^A, c, m) = c\ m(\texttt{void})$.
  - We also know that each $\mathcal{M}(P^A, c, unlock_i) = \texttt{void}\ unlock_i(\texttt{void})$.
  - We can now show that $P^S, [m\texttt{\_x} \mapsto \texttt{void}, \texttt{this} \mapsto c] \vdash \mathbf{e}^A : c$ because all the $unlock_i$ calls can be typed to `void`, and thus their composition can be typed to `void`, but the composition of the calls and `this`returns c.

For all asynchronous chords $(\{m'_1 \ldots m'_n\}, \mathbf{e}^A) \in \mathcal{ACh}s(P^A, c)$, there is only one case, so we know that:

- $\delta(l) = (c, \_, unlock, \{(m_1, m'_1) \ldots (m_n, m'_n)\}, \mathbf{e}^S)$ and from the definition of φ_{SA}, that leads to $\mathcal{M}(P^A, c, m'_i) = \texttt{void}\ m'_i(t_i)$ where $\mathcal{M}(P^S, c, m_i) = \texttt{void}\ m_i(t_i)$ and $\mathcal{M}(P^S, c, unlock) = \texttt{void}\ unlock(\texttt{void})$.
- $\mathbf{e}^A = \texttt{this}.unlock(\texttt{voidval}); \mathbb{C}\sigma(\mathbf{e}^S)$ where $\sigma = [m'_1\texttt{\_x}/m_1\texttt{\_x} \ldots m'_n\texttt{\_x}/m_n\texttt{\_x}]$.

- From $P^S \vdash \delta$ we know that $[\texttt{spawn}\ \mathbf{e}^S]^l$ is a subterm of $\mathbf{e}^S_{ch}$ where $(\{m_2 \ldots m_n\}, \mathbf{e}^S_{ch}) \in \mathcal{SCHS}(P^S, c, m_1)$ and from (WFClass) we know that $P^S, \Gamma \vdash \mathbf{e}^S_{ch} : t$ ($\Gamma = [m_1\texttt{\_x} \mapsto t_1 \ldots m_n\texttt{\_x} \mapsto t_n, \texttt{this} \mapsto c]$) for some type t, which implies $P^S, \Gamma \vdash [\texttt{spawn}\ \mathbf{e}^S]^l : \texttt{void}$ since all type rules require the subterms to be well-typed. It immediately follows that $P^S, \Gamma \vdash \mathbf{e}^S : \texttt{void}$.

- Using the above and lemma 6.3.3, we know that $P^A, \Gamma \vdash \mathbb{C}(\mathbf{e}^S) : \texttt{void}$ but we need to prove $P^A, \sigma(\Gamma) \vdash \texttt{this}.\mathit{unlock}(\texttt{voidval}); \sigma(\mathbb{C}(\mathbf{e}^S)) : \texttt{void}$. This follows using the previously determined $\mathcal{M}(P^S, c, \mathit{unlock}) = \texttt{void}\ \mathit{unlock}(\texttt{void})$ and lemma 6.3.4.

The above is sufficient to prove $P^S \vdash c$. □

Theorem 6.3.6 $\vdash P^S \Longrightarrow \vdash P^A$.

Proof:

$$
\begin{array}{lll}
\vdash P^S & \Longrightarrow \forall c. \mathcal{Sup}(P^S, c) \neq \mathcal{Udf} \Rightarrow P^S \vdash c & \text{(WFProg)} \\
& \Longrightarrow \forall c. \mathcal{Sup}(P^A, c) \neq \mathcal{Udf} \Rightarrow P^S \vdash c & \text{(Def } \varphi_{AS}\text{)} \\
& \Longrightarrow \forall c. \mathcal{Sup}(P^A, c) \neq \mathcal{Udf} \Rightarrow P^A \vdash c & \text{(Lemma 6.3.5)} \\
& \Longrightarrow \vdash P^A & \text{(WFProg)}
\end{array}
$$

Chapter 7

$\mathcal{L}^J$ is at most as expressive as $\mathcal{L}^{S+}$

Up to this point, we have only considered formalisations that use chords for synchronisation. Now we define a language with monitors, $\mathcal{L}^J$. This formalisation is so-called because it is loosely based around Java. It has similar object-oriented features, and it associates a mutex with each object. The constructs that implement condition variables are also based around objects.

However, unlike Java, we do not have a `synchronized` construct, opting instead for the "unstructured" synchronisation where an object ι has a pair of interactions, `lock` ι and `unlock` ι. We use non-re-entrant locks. Unlike previous formalisations, $\mathcal{L}^J$ needs field members that represent state. A complete comparison with Java, including justification of these differences, is below.

In this chapter we show that $\mathcal{L}^J$ is at least as expressive as a new formalisation, $\mathcal{L}^{S+}$. This new formalisation extends $\mathcal{L}^S$ with fields and constructs for dealing with integers. It is easier for us to encode $\mathcal{L}^J$ programs into a formalisation with a `spawn` construct, (rather than asynchronous chords), because $\mathcal{L}^J$ also has a `spawn` construct.

We need not have extended $\mathcal{L}^S$ with fields, since we know from [4] that the fields of $\mathcal{L}^J$ could be represented with the queues of $\mathcal{L}^{S+}$. However, we prefer not to include this in our translation because it would not prove anything new, and it distracts from the translation of the synchronisation constructs which is the main aim of this chapter. The integers are required because the implementation of condition variables with chords requires us to keep a count of the number of waiting threads.

Essentially, we show that for each program design that uses the semantics of monitors, there is an equivalent design that uses synchronous chords for the same effect.

7.1 Definition of $\mathcal{L}^J$

For modelling monitors, we use a formalisation that I developed independently, but which turned out to be very similar to that which is presented in [1]. A full comparison will be given below.

We formalise $\mathcal{L}^J$, an object-oriented language like $\mathcal{L}^S$ but with monitors and conventional methods instead of chords. The design of $\mathcal{L}^J$ is inspired by Java, but instead of having a `synchronized` keyword, we instead use `lock()` and `unlock()` methods like POSIX.

7.1.1 Comparison with the formalisation in [1]

The formalism in [1] is a far more complete model of the Java language, as it was intended to not just model the synchronisation in Java, but also the language features that enable distributed applications, such as remote method invocation. If we consider only the synchronisation semantics from the [1] formalism, there are still some differences:

- The [1] formalism models the "structured" synchronisation that is due to the Java language's `synchronized` keyword.
- The [1] formalism models re-entrant mutexes, by storing an integer counter within the run-time state.
- At a technical level, the [1] formalism uses a number of pre-defined functions and predicates in its rules which our simpler semantics avoids doing.

7.1.2 Comparison with real languages such as Java

The purpose of the formalism is to model the behaviour of monitors in real programming languages, and aside from these features only the essential constructs are present in the language. Even some aspects of monitors have been omitted, where it was decided that they did not contribute the expressiveness of the synchronisation, or where their contribution was obvious.

We model unstructured, non-re-entrant mutexes, although Java uses more sophisticated mutexes, and POSIX supports a range of different kinds of mutexes. There are two reasons for our simplification:

Firstly, the unstructured non-re-entrant mutex is naturally expressed by the semantics of chords. Suppose we can prove that chords are as expressive as this kind of mutex. We can then compare chords to other kinds of mutex by simply comparing unstructured non-re-entrant mutexes to these other kinds of mutex. Comparing mutexes to other kinds of mutexes is much simpler than comparing chords to other kinds of mutexes.

Secondly, a language that has structured synchronisation is as expressive as a language that has unstructured synchronisation. To support this, we give the following formally unsubstantiated, but still reasonable claims:

- Programs using the `synchronized` keyword can be converted to programs using unstructured synchronisation by calling `lock()` and `unlock()` respectively before and after the synchronized block.
- The functionality of `lock()` and `unlock()` methods can be implemented on top of the `synchronized` keyword using condition variables to do the actual blocking of threads as described in section 2.2.

Likewise, whether the language has re-entrant or non-re-entrant mutexes is a non-issue: Firstly re-entrant mutexes can be implemented on top of non-re-entrant mutexes as seen in

section 2.2 (assuming some mechanism for identifying threads is present in the language[1]) and this implementation can be used as a drop-in replacement for re-entrant mutexes (i.e. without changing the program structure). Secondly, if a program is written using non-re-entrant threads, and it does not deadlock (correct programs should not deadlock), then it never attempts to lock a mutex when it already has that lock, and thus the program does not distinguish the behaviour of re-entrant and non-re-entrant locks, thus re-entrant locks are a drop-in replacement for non-re-entrant locks.

In none of our formalisations do we model any notion of thread identity. In Java this is represented by instances and static members of the Thread object. In POSIX, there is a structure for holding a thread identity, and functions such as `pthread_self` for identifying the current thread. We feel these were not interesting enough to include in the formalism, since they do not directly contribute to synchronisation itself, although they are required for certain synchronisation algorithms, such as implementing re-entrant locks.

Both Java and POSIX have features for doing a `wait()` call with a timeout. This might be required for certain programs but we feel it is not central to the issue of synchronisation. There are also mechanisms for interrupting threads that are in a waiting state, and exceptions that are thrown in this event. Again, we do not model these as we feel they are more specialist features and we want the semantics to be as simple as possible.

Java 1.5 and POSIX also give the programmer the ability to test the availability of a mutex, i.e. a call like `lock()` that does not block, but instead returns a success / failure status. We consider this a specialist feature.

Java 1.5's `Lock` class allows the use of mutexes that are not associated with an object, and likewise its `Condition` class allows multiple condition variables to be associated with a single lock. We opt for the simpler technique from previous versions of Java, where each object has a single mutex, and a single associated condition variable. Thus the constructs for locking mutexes and interacting with condition variables can refer to an object, without having to represent a separate notion of mutex and condition variable.

We believe decoupling mutexes from objects and condition variables from mutexes does not add to the expressiveness of the language because a set of condition variables can be represented with a single condition variable if a shared memory message is 'sent' to all the waiting threads, telling it exactly which event has occurred. Likewise, it is always possible to create a dummy object solely for the purpose of using its mutex.

7.1.3 Guided tour of $\mathcal{L}^J$

Because there are many similarities between $\mathcal{L}^J$ and $\mathcal{L}^S$, we only explain the aspects of the semantics which are different. Chapter 4 explains the technical nature of $\mathcal{L}^A$ and $\mathcal{L}^S$, and thus the same information applies here. The definition of $\mathcal{L}^J$ is given in figure 7.1, figure 7.2, and figure 7.3.

[1]If we do not have some notion of thread identity, and being able to compare the current thread with some recorded notion of thread identity, then we can pass information describing what locks we currently have into each method, by including an extra argument, and then make run-time decisions about whether or not to lock. This is necessary because the stack is the only area of storage which is owned by a single thread. Having to do this would require massive changes to any program structure however, since this information would need to be maintained, and added to all method calls. It is similar to the encoding of exceptions with an augmented return value.

Programs

The first difference is that a program $P^J \in \mathcal{L}^J$ does not have synchronous chords, instead there is a mapping from methods identifiers to method bodies. Thus, each method can have at most one body. The program tuple also contains a mapping that represents which fields are defined in each class.

In $\mathcal{L}^J$, we only need to consider fields of class type, since the only other type is void, and void fields add nothing to the expressiveness of the language. Since their value is always `voidval`, we can replace $\iota.f$ with `voidval` and $\iota.f := \texttt{voidval}$ with `voidval`.

Source expressions contain two new groups of constructs, constructs for synchronisation (`lock` **e**, `unlock` **e**, `wait` **e**, `notify` **e**, and `notifyAll` **e**) and constructs for field lookup and assignment (**e**.f and **e**.fL = **e**). $\mathcal{L}^J$ method bodies reference the single argument variable with the constant `x`.

The synchronisation constructs `lock` **e** and `unlock` **e** represent the methods for locking and unlocking a mutex. Every instance has its own mutex, the sub-expression resolves to the address of the object that contains the mutex we want to control. The constructs `wait` **e**, `notify` **e**, and `notifyAll` **e**, represent the public methods of class `Object` in Java, that give the programmer access to the condition variables in the language. We could have implemented these as methods in our formalisation, but it seemed clearer to introduce them as language constructs, rather than creating special cases of the semantics rule for method calls.

Execution

The run-time heap is similar to that in $\mathcal{L}^S$ although instead of queues, we have fields. Each field can store a single value, so we do not need a $Id_f \rightarrow Multiset(Val)$ definition. Run-time expressions incorporate the synchronisation constructs descried above, but also `locked` ι e and `waiting` ι.

If a run-time expression, or thread, e contains `locked` ι e' anywhere within its structure, then that thread has the lock on the instance ι. I.e. if $e = E_1[\texttt{locked}\ \iota\ e']$ then e has the lock on ι. This allows us to associate this information with an individual thread, which would be difficult it do if we were, for example, to place the information in the heap.

The construct `waiting` ι does not feature in any so it will not get "stepped over" like v in $v; e$. A thread $e = E[\texttt{waiting}\ \iota]$ is waiting on the mutex within the instance ι, as described in section 2.2. This is only meaningful during run-time, so `waiting` was not included in the source expressions of $\mathcal{L}^J$.

The rule (Run) does not factor out the contexts from its sub-rules (New), (Inv), (Unlock), (Wait), (Fld), and (FldAss), as was the case in $\mathcal{L}^A$ and $\mathcal{L}^S$, because the rules (Unlock) and (Wait) need to use this information, as will be discussed below. Aside from this detail, the (New) rule is the same as in $\mathcal{L}^S$ except that it sets the field values to `null` and doesn't need to deal with the queues.

The rule (Inv) takes the place of (InvS) and (InvA). With no queues, and only one argument per invocation, the rule is much simpler. There is also only one possible method body. Method invocation does not affect the heap in $\mathcal{L}^J$. The only operations that affect the heap are field operations and object construction.

The execution of the `lock` ι statement should only proceed when no other threads

have the lock, as described in section 2.2. Thus there is only one semantics rule that processes `lock` ι and that is (Lock). The rule only applies when no other threads have the lock, which we implement by testing all the other threads and then separately testing the thread that is actually executing `lock` ι. Because we require the thread which is doing the locking to not have the lock, this is a non-re-entrant mutex. If no thread has the lock then we represent the obtaining of the lock by placing `locked` ι at the root of the run-time expression that models the thread. The rule (Unlock) simply removes this information from the thread in question.

The rule (Wait) will only execute when it has the lock on the object that is being waited upon, this is a necessary feature of condition variables and is mentioned in section 2.2. Since a thread can only get the lock on an object by executing `lock` ι *itself*, this means that if a thread attempts to execute `wait` ι without the lock then it will block forever. This is like de-referencing a null pointer, and in Java an exception is raised (the Java API documentation for `Object` explicitly instructs us to make sure we have the lock on an object before calling `wait()`). Therefore valid programs never call `wait` ι without the lock on that object. Aside from this detail, `wait` ι simply gives up the lock on the object and enters the waiting state.

The rule (Notify) affects not only the thread calling `notify` ι (the expression returns `voidval`) but also another arbitrary thread in the system that is waiting on the same object. The rule also ensures that the thread calling `notify` ι has the lock on ι, although it does not give the lock up like (Wait) does. When threads are woken up, they change from $E[$`waiting` $\iota]$ to $E[$`lock` $\iota]$ to model the fact that waking threads must get back the lock they gave up before proceeding.

The rule (NotifyNone) applies if there are no threads to notify, i.e. it complements (Notify). The premise of the rule ensures that no threads are waiting, and reduces the `notify` ι to `voidval`. Like the other notification rules, it will not apply unless the notifying thread has the lock.

The rule (NotifyAll) is very similar to (Notify) but affects all the threads in the system that are waiting on the object in question. Every thread is either left unchanged if they were not waiting on ι (the "otherwise" case) or converted to $E[$`lock` $\iota]$ if they were waiting.

The field rules (Fld) and (FldAss) simply look up and re-write the function that represents the field values within the relevant instance.

Well-formedness

The only difference in the well-formedness relation of $\mathcal{L}^J$ when compared to $\mathcal{L}^S$ is that we have to ensure that fields are preserved exactly over inheritance. Instead of checking the types of the chord bodies, we have to check method bodies instead, but the idea is the same.

The types of the synchronisation constructs are all the same. All the constructs return `voidval` in their own thread, and act on an object, so the type rule for all of them merely ensures that the sub-expression is of some class type, and types the construct itself to `void`. This makes sense, because they are triggering an external effect in other threads, rather than computing with data.

The type rules for the field lookup and assignment operations use the mapping in the program tuple to determine the type that a field's contents should be.

Programs:

$$
\begin{array}{rcll}
P^J \in \mathcal{L}^J & = & Id_c \times Id_m \to Methsig & \text{(Type signatures)} \\
 & \times & Id_c \times Id_m \to SrcExpr & \text{(Method bodies)} \\
 & \times & Id_c \times Id_f \to Id_c & \text{(Fields)} \\
 & \times & Id_c \to Id_c & \text{(Superclass)} \\
Methsig & ::= & t\ m(t) & \\
t \in Types & ::= & c \mid \texttt{void} & \\
\mathbf{e} \in SrcExpr & ::= & \texttt{null} \mid \texttt{voidval} \mid \texttt{this} \mid \texttt{x} \mid \texttt{new}\ c \mid \mathbf{e}.m(\mathbf{e}) & \\
 & \mid & \mathbf{e}\ ;\ \mathbf{e} \mid \texttt{spawn}\ \mathbf{e} \mid \texttt{lock}\ \mathbf{e} \mid \texttt{unlock}\ \mathbf{e} \mid \texttt{wait}\ \mathbf{e} & \\
 & \mid & \texttt{notify}\ \mathbf{e} \mid \texttt{notifyAll}\ \mathbf{e} \mid \mathbf{e}.f \mid \mathbf{e}.f := \mathbf{e} & \\
m \in Id_m & & c \in Id_c \qquad f \in Id_f &
\end{array}
$$

$$
\mathcal{M}(P^J, c, m) = P^J{\downarrow_1}(c, m) \qquad \mathcal{M}\mathit{eth}(P^J, c, m) = P^J{\downarrow_2}(c, m)
$$
$$
\mathcal{F}(P^J, c) = P^J{\downarrow_3}(c) \qquad \mathcal{S}\mathit{up}(P^J, c) = P^J{\downarrow_4}(c)
$$

Runtime objects:

$$
\begin{array}{rcl}
h \in Heap & = & \mathbb{N} \to (Id_c \times (Id_f \to Val)) \\
e \in RunExpr & ::= & v \mid \texttt{new}\ c \mid e.m(e) \mid e\ ;\ e \mid \texttt{spawn}\ e \\
 & \mid & \texttt{lock}\ e \mid \texttt{unlock}\ e \mid \texttt{wait}\ e \mid \texttt{notify}\ e \\
 & \mid & \texttt{notifyAll}\ e \mid e.f \mid e.f := e \\
 & \mid & \texttt{locked}\ \iota\ e \mid \texttt{waiting}\ \iota \\
v \in Val & ::= & \texttt{null} \mid \texttt{voidval} \mid \iota \\
\iota \in \mathbb{N} & & \text{(Addresses)} \\
E[\cdot] & ::= & E[\cdot].m(e) \mid \iota.m(E[\cdot]) \mid E[\cdot]\ ;\ e \mid v\ ;\ E[\cdot] \\
 & \mid & \texttt{lock}\ E[\cdot] \mid \texttt{unlock}\ E[\cdot] \mid \texttt{wait}\ E[\cdot] \\
 & \mid & \texttt{notify}\ E[\cdot] \mid \texttt{notifyAll}\ E[\cdot] \mid \texttt{locked}\ \iota\ E[\cdot] \\
 & \mid & E[\cdot].f \mid E[\cdot].f := e \mid \iota.f := E[\cdot]
\end{array}
$$

Figure 7.1: Syntax of $\mathcal{L}^J$.

$$\frac{P^J \vdash e, h \rightsquigarrow e', h'}{P^J \vdash e_1 \dots e_n, e, h \rightsquigarrow e_1 \dots e_n, e', h'} \quad \text{(Run)}$$

$$\frac{h(\iota) = \mathcal{U}df}{P^J \vdash E[\texttt{new } c], h \rightsquigarrow E[\iota], h[\iota \mapsto [\![c \| \lambda f.\texttt{null}]\!]]} \quad \text{(New)}$$

$$\frac{h(\iota) = [\![c \| \_]\!], \qquad \mathbf{e} = \mathcal{M}eth(P, c, m)}{P^J \vdash E[\iota.m(v)], h \rightsquigarrow E[\mathbf{e}[v/\texttt{x},\ \iota/\texttt{this}]], h} \quad \text{(Inv)}$$

$$\frac{\begin{array}{c}\forall i \in \{1 \dots n\}. \nexists E'. e_i = E'[\texttt{locked } \iota\ \_] \\ \nexists E'. E[\texttt{lock } \iota] = E'[\texttt{locked } \iota\ \_]\end{array}}{P^J \vdash e_1 \dots e_n, E[\texttt{lock } \iota], h \rightsquigarrow e_1 \dots e_n, \texttt{locked } \iota\ E[\texttt{voidval}], h} \quad \text{(Lock)}$$

$$\frac{}{P^J \vdash E_1[\texttt{locked } \iota\ E_2[\texttt{unlock } \iota]]], h \rightsquigarrow E_1[E_2[\texttt{voidval}]], h} \quad \text{(Unlock)}$$

$$\frac{}{P^J \vdash E_1[\texttt{locked } \iota\ E_2[\texttt{wait } \iota]]], h \rightsquigarrow E_1[E_2[\texttt{waiting } \iota]], h} \quad \text{(Wait)}$$

$$\frac{}{\begin{array}{r}P^J \vdash e_1 \dots e_n, E_1[\texttt{waiting } \iota], E_2[\texttt{locked } \iota\ E_3[\texttt{notify } \iota]], h \rightsquigarrow \\ e_1 \dots e_n, E_1[\texttt{lock } \iota], E_2[\texttt{locked } \iota\ E_3[\texttt{voidval}]], h\end{array}} \quad \text{(Notify)}$$

$$\frac{\forall i \in \{1 \dots n\}. \nexists E'. e_i = E'[\texttt{waiting } \iota]}{\begin{array}{r}P^J \vdash e_1 \dots e_n, E_1[\texttt{locked } \iota\ E_2[\texttt{notify } \iota]], h \rightsquigarrow \\ e_1 \dots e_n, E_1[\texttt{locked } \iota\ E_2[\texttt{voidval}]], h\end{array}} \quad \text{(NotifyNone)}$$

$$\frac{\forall i \in \{1 \dots n\}. e'_i = \begin{cases} E'[\texttt{lock } \iota] & \text{if } e_i = E'[\texttt{waiting } \iota] \\ e_i & \text{otherwise} \end{cases}}{\begin{array}{r}P^J \vdash e_1 \dots e_n, E_1[\texttt{locked } \iota\ E_2[\texttt{notifyAll } \iota]], h \rightsquigarrow \\ e'_1 \dots e'_n, E_1[\texttt{locked } \iota\ E_2[\texttt{voidval}]], h\end{array}} \quad \text{(NotifyAll)}$$

$$\frac{}{P^J \vdash e_1 \dots e_n, E[\texttt{spawn } e], h \rightsquigarrow e_1 \dots e_n, E[\texttt{voidval}], e, h} \quad \text{(Spawn)}$$

$$\frac{h(\iota) = [\![c \| fs]\!]}{P^J \vdash E[\iota.f], h \rightsquigarrow E[fs(f)], h} \quad \text{(Fld)}$$

$$\frac{h(\iota) = [\![c \| fs]\!]}{P^J \vdash E[\iota.f := v], h \rightsquigarrow E[v], h[\iota \mapsto [\![c \| fs[f \mapsto v]]\!]]} \quad \text{(FldAss)}$$

Figure 7.2: Semantics of $\mathcal{L}^J$.

Well-formedness:

$$\frac{\forall c.\mathcal{S}up(P^J, c) \neq \mathcal{U}df \implies P^J \vdash c}{\vdash P^J} \quad \text{(WFProg)}$$

$$\frac{\begin{array}{l} P^J \vdash \mathcal{S}up(P^J, c) \diamond_{cl} \\ \forall m.\mathcal{M}(P^J, \mathcal{S}up(P^J, c), m) = t_r\ m(t_a) \implies \exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a.\mathcal{M}(P^J, c, m) = t'_r\ m(t'_a) \\ \forall m, \mathbf{e} = \mathcal{M}eth(P^J, c, m) \\ \qquad\qquad \mathcal{M}(P^J, c, m) = t_r\ m(t_a), \\ \qquad\qquad P^J, [\mathtt{x} \mapsto t_a, \mathtt{this} \mapsto c] \vdash \mathbf{e} : t_r \\ \forall f.\mathcal{F}(P^J, c, f) = c' \implies P^J \vdash c' \diamond_{cl} \\ \forall f.\mathcal{F}(P^J, \mathcal{S}up(P^J, c), f) = t \implies \mathcal{F}(P^J, c, f) = t \end{array}}{P^J \vdash c} \quad \text{(WFClass)}$$

$$\frac{\mathcal{S}up(P^J, c) \neq \mathcal{U}df \quad \vee \quad c = \mathtt{Object}}{P^J \vdash c \diamond_{cl}} \quad \text{(IsClass)}$$

Source type rules: (for (STLock), (STUnlock), (STNotify) and (STNotifyAll) see (STWait))

$$\frac{v \in \{\mathtt{this}, \mathtt{x}\}}{P^J, \Gamma \vdash v : \Gamma(v)} \quad \text{(STVar)} \qquad \frac{P^J \vdash c \diamond_{cl}}{P^J, \Gamma \vdash \mathtt{null} : c} \quad \text{(STNull)}$$

$$\frac{}{P^J, \Gamma \vdash \mathtt{voidval} : \mathtt{void}} \quad \text{(STVoid)} \qquad \frac{P^J \vdash c \diamond_{cl}}{P^J, \Gamma \vdash \mathtt{new}\ c : c} \quad \text{(STNew)}$$

$$\frac{\begin{array}{c} P^J, \Gamma \vdash \mathbf{e}_1 : c \\ P^J, \Gamma \vdash \mathbf{e}_2 : t \\ \mathcal{M}(P^J, c, m) = t_r\ m(t) \end{array}}{P^J, \Gamma \vdash \mathbf{e}_1.m(\mathbf{e}_2) : t_r} \quad \text{(STInv)} \qquad \frac{\begin{array}{c} P, \Gamma \vdash \mathbf{e}_1 : t_1 \\ P, \Gamma \vdash \mathbf{e}_2 : t_2 \end{array}}{P, \Gamma \vdash \mathbf{e}_1\ ;\ \mathbf{e}_2 : t_2} \quad \text{(STSeq)}$$

$$\frac{\begin{array}{c} P^J, \Gamma \vdash \mathbf{e} : c \\ P^J, \Gamma \vdash \mathcal{S}up(P^J, c) = c' \end{array}}{P^J, \Gamma \vdash \mathbf{e} : c'} \quad \text{(STSub)} \qquad \frac{P^J, \Gamma \vdash \mathbf{e} : \mathtt{void}}{P^J, \Gamma \vdash \mathtt{spawn}\ \mathbf{e} : \mathtt{void}} \quad \text{(STSpawn)}$$

$$\frac{P^J, \Gamma \vdash \mathbf{e} : c}{P^J, \Gamma \vdash \mathtt{wait}\ \mathbf{e} : \mathtt{void}} \quad \text{(STWait)}$$

$$\frac{\begin{array}{c} P^J, \Gamma \vdash \mathbf{e} : c \\ \mathcal{F}(P^J, c, f) = t \end{array}}{P^J, \Gamma \vdash \mathbf{e}.f : t} \quad \text{(STFld)} \qquad \frac{\begin{array}{c} P^J, \Gamma \vdash \mathbf{e}_1 : c \\ \mathcal{F}(P^J, c, f) = t \\ P^J, \Gamma \vdash \mathbf{e}_2 : t \end{array}}{P^J, \Gamma \vdash \mathbf{e}_1.f := \mathbf{e}_2 : t} \quad \text{(STAss)}$$

Figure 7.3: Rules for well-formed $\mathcal{L}^J$ programs.

7.1.4 Invariants of execution in $\mathcal{L}^J$ programs

These invariants illustrate the constraints on the state of an executing $\mathcal{L}^J$ program. The first invariant is due to redundancies in the formalisation. The other two invariants are due to the fundamental nature of synchronisation with monitors, and the constraints that this imposes on the execution.

- Firstly, when a thread gets the lock on an object, `locked` ... is placed at the root of the run-time expression. This means that the following property on a state $s : Multiset(RunExpr) \times Heap$ will always be maintained over execution:

  $\forall e \in s{\downarrow_1} \,.\, \exists n{\geq}0, e' \,.\, e = \texttt{locked}\ \iota_1\ \texttt{locked}\ \iota_2\ \ldots \texttt{locked}\ \iota_n\ e',\ \nexists E \,.\, e' = E[\texttt{locked}\ \_\ \_]$

  In words – the `locked` ι expressions will be together at the root of all the run-time expressions that model threads. For each thread, $\iota_1 \ldots \iota_n$ is the set of objects on which the thread has the lock.

- Secondly, a thread gives up the lock on an object when it is waiting, so we always know the following property will be maintained over execution.

  $\forall e \in s{\downarrow_1} \,.\, e = E[\texttt{waiting}\ \iota] \implies \nexists E' \,.\, e = E'[\texttt{locked}\ \iota\ \_]$

  In words – If a thread is waiting on an object, it does not have the lock on that object.

- Finally, the property of mutual exclusion is that no two threads have the lock on the same object at any one time. Also, a thread can only have the lock once.

  $\forall e_1 \in s{\downarrow_1}, e_2 \in s{\downarrow_2}, E_1, E_2 \,.$
  $(e_1 = E_1[\texttt{locked}\ \iota\ \_] \wedge e_2 = E_2[\texttt{locked}\ \iota\ \_]) \implies (e_1 = e_2 \wedge E_1 = E_2)$

  In words – If two threads have the lock on the some object ι, then those two threads are in fact the same thread, and also the lock is in the same place of the syntax tree for both of them.

7.2 Definition of $\mathcal{L}^{S+}$

The language $\mathcal{L}^{S+}$ is an extension of $\mathcal{L}^S$ that adds fields and integers. The fields do not add any expressiveness to the language, as it was shown in [4] that there is an encoding of fields within the system of queues used in the semantics of chords, and this encoding happens to be structure preserving by our definition.

7.2.1 Guided tour of $\mathcal{L}^{S+}$

The additional language constructs are `0`, `inc` **e**, `dec` **e**, and `nonzero` e e. These are the constant integer zero, the predecessor and successor functions, and a kind of conditional statement, respectively.

We extend the set of types with `int`, and note that we must now allow fields of both class type (Id_c) and `int` type. When making new objects with the (New) rule, we use the $Init(P^{S+}, c, f)$ predicate to obtain the correct initial value for the fields, since we can no-longer blindly use `null`.

The rules that act below the (Run) rule, i.e. (New), (InvA), (FldAss), (Inc), (Dec), and (Nonzero), act on the whole runtime expression, including the context, which is different from the way that the semantics rules of $\mathcal{L}^S$ are designed. In $\mathcal{L}^S$ we factored out the context $E[\cdot]$ from all the "single thread" rules into the (Run) rule, since it was duplicated in each case. With $\mathcal{L}^J$ however, this was no longer appropriate since some rules needed to refer to the thread's lock status, which was embedded within the context of the run-time expression. Since $\mathcal{L}^{S+}$ is to be compared to $\mathcal{L}^J$ we make the style of the $\mathcal{L}^{S+}$ rules the same as the style of $\mathcal{L}^J$ rules.

The (InvA), (InvS), and (Spawn) rules are otherwise taken straight from $\mathcal{L}^S$. The field rules are standard, and the same as in $\mathcal{L}^J$.

The rules (Inc) and (Dec) are straightforward, they define the usual successor and predecessor functions on the integers. The definition of contexts allows the e in `inc` e to be evaluated into an integer, and the (Inc) rule then adds one to this integer. It does not change any state, like e.g. the `++` syntax in the C programming language.

The rule (Nonzero) is a conditional construct. Depending on its first argument, executing `nonzero` e_1 e_2 will either execute e_2 or not. If e_1 is zero, the expression reduces to `voidval`, thus discarding e_2, otherwise it reduces to e_2, and the subsequent steps of execution will be the execution of e_2. The typing rule for `nonzero` e_1 e_2 requires e_2 to be `void`, because the return value must be the same regardless of whether e_1 was zero (and thus the return value is `voidval`) or not.

Programs:

$$
\begin{array}{rcll}
P^{S+} \in \mathcal{L}^{S+} & = & Id_c \times Id_m \to Methsig & \text{(Type signatures)} \\
& \times & Id_c \times Id_m \to \mathbb{P}(Chord) & \text{(Synchronous chords)} \\
& \times & Id_c \times Id_f \to (Id_c \cup \{\texttt{int}\}) & \text{(Fields)} \\
& \times & Id_c \to Id_c & \text{(Superclass)} \\
Methsig & ::= & t\ m(t) & \\
t \in Types & ::= & c \mid \texttt{void} \mid \texttt{int} & \\
Chord & = & \mathbb{P}(Id_m) \times SrcExpr & \\
\mathbf{e} \in SrcExpr & ::= & \texttt{null} \mid \texttt{voidval} \mid \texttt{this} \mid m\_\texttt{x} \mid \texttt{new}\ c \mid \mathbf{e}.m(\mathbf{e}) \mid \mathbf{e}\ ;\ \mathbf{e} \mid \texttt{spawn}\ \mathbf{e} & \\
& \mid & \mathbf{e}.f \mid \mathbf{e}.f := \mathbf{e} \mid \texttt{0} \mid \texttt{inc}\ \mathbf{e} \mid \texttt{dec}\ \mathbf{e} \mid \texttt{nonzero}\ \mathbf{e}\ \mathbf{e} & \\
m \in Id_m & & c \in Id_c \qquad f \in Id_f &
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{M}(P^{S+}, c, m) = P^{S+}{\downarrow_1}(c, m) & \mathcal{SCHs}(P^{S+}, c, m) = P^{S+}{\downarrow_2}(c, m) \\
\mathcal{F}(P^{S+}, f) = P^{S+}{\downarrow_3}(c, f) & \mathcal{Sup}(P^{S+}, c) = P^{S+}{\downarrow_4}(c)
\end{array}
$$

$$
\mathcal{Q}(P^S, c) = \bigcup\{\ ch{\downarrow_1} \mid \exists m.ch \in \mathcal{SCHs}(P^{S+}, c, m)\ \}
$$

Runtime objects:

$$
\begin{array}{rcl}
h \in Heap & = & \mathbb{N} \to (Id_c \times (Id_m \to Multiset(Val)) \times (Id_f \to Val)) \\
e \in RunExpr & ::= & v \mid \texttt{new}\ c \mid e.m(e) \mid e\ ;\ e \mid \texttt{spawn}\ e \mid e.f \mid e.f := e \\
& & \texttt{inc}\ e \mid \texttt{dec}\ e \mid \texttt{nonzero}\ e\ e \\
v \in Val & ::= & \texttt{null} \mid \texttt{voidval} \mid \iota \mid z \\
\iota \in \mathbb{N} & & z \in \mathbb{Z} \\
E[\cdot] & ::= & E[\cdot].m(e) \mid \iota.m(E[\cdot]) \mid E[\cdot]\ ;\ e \mid v\ ;\ E[\cdot] \\
& \mid & E[\cdot].f \mid E[\cdot].f := e \mid \iota.f := E[\cdot] \\
& \mid & \texttt{nonzero}\ E[\cdot]\ e \mid \texttt{inc}\ E[\cdot] \mid \texttt{dec}\ E[\cdot]
\end{array}
$$

Initial values of fields are defined as follows:

$$
Init(P^{S+}, c, f) = \begin{cases} \texttt{null} & \text{if } \mathcal{F}(P^{S+}, c, f) \in Id_c \\ \texttt{0} & \text{if } \mathcal{F}(P^{S+}, c, f) = \texttt{int} \end{cases}
$$

Figure 7.4: Syntax of $\mathcal{L}^{S+}$.

$$\frac{P^{S+} \vdash e, h \rightsquigarrow e', h'}{P^{S+} \vdash e_1 \ldots e_n, e, h \rightsquigarrow e_1 \ldots e_n, e', h'} \quad \text{(Run)}$$

$$\frac{h(\iota) = \mathcal{U}df}{P^{S+} \vdash E[\texttt{new } c], h \rightsquigarrow E[\iota], h[\iota \mapsto [\![c \| \lambda m.\emptyset \| \lambda f.\mathit{Init}(P^{S+}, c, f)]\!]]} \quad \text{(New)}$$

$$\frac{h(\iota) = [\![c \| qs \| fs]\!], \qquad m \in \mathcal{Q}(P^{S+}, c)}{P^{S+} \vdash E[\iota.m(v)], h \rightsquigarrow E[\texttt{voidval}], h[\iota \mapsto [\![c \| qs[m \mapsto qs(m) \uplus \{v\}] \| fs]\!]]} \quad \text{(InvA)}$$

$$\frac{\begin{array}{c} h(\iota) = [\![c \| qs \| fs]\!], \qquad (\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{SC}\mathit{hs}(P^{S+}, c, m) \\ \forall i \in \{1 \ldots n\}.qs(m_i) = \{v_i\} \uplus q_i \\ h' = h[\iota \mapsto [\![c \| qs[m_1 \mapsto q_1 \ldots m_n \mapsto q_n] \| fs]\!]] \end{array}}{P^{S+} \vdash E[\iota.m(v)], h \rightsquigarrow E[\mathbf{e}[v_1/m_1\texttt{\_x} \ \ldots \ v_n/m_n\texttt{\_x}, \ v/m\texttt{\_x}, \ \iota/\texttt{this}]], h'} \quad \text{(InvS)}$$

$$\frac{}{P^{S+} \vdash e_1 \ldots e_n, E[\texttt{spawn } e], h \rightsquigarrow e_1 \ldots e_n, E[\texttt{voidval}], e, h} \quad \text{(Spawn)}$$

$$\frac{h(\iota) = [\![c \| qs \| fs]\!]}{P^{S+} \vdash E[\iota.f], h \rightsquigarrow E[fs(f)], h} \quad \text{(Fld)}$$

$$\frac{h(\iota) = [\![c \| qs \| fs]\!]}{P^{S+} \vdash E[\iota.f := v], h \rightsquigarrow E[v], h[\iota \mapsto [\![c \| qs \| fs[f \mapsto v]]\!]]} \quad \text{(FldAss)}$$

$$\frac{}{P^{S+} \vdash E[\texttt{inc } z], h \rightsquigarrow E[z+1], h} \quad \text{(Inc)}$$

$$\frac{}{P^{S+} \vdash E[\texttt{dec } z], h \rightsquigarrow E[z-1], h} \quad \text{(Dec)}$$

$$\frac{e' = \begin{cases} \texttt{voidval} & \text{if } z = 0 \\ e & \text{otherwise} \end{cases}}{P^{S+} \vdash E[\texttt{nonzero } z \ e], h \rightsquigarrow E[e'], h} \quad \text{(Nonzero)}$$

Figure 7.5: Semantics of $\mathcal{L}^{S+}$.

Well-formedness:

$$\frac{\forall c.\mathcal{S}up(P^{S+},c) \neq \mathcal{U}df \implies P^{S+} \vdash c}{\vdash P^{S+}} \quad \text{(WFProg)}$$

$$\frac{\begin{array}{l} P^{S+} \vdash \mathcal{S}up(P^{S+},c)\ \diamond_{cl} \\ \forall m.\mathcal{M}(P^{S+},\mathcal{S}up(P^{S+},c),m) = t_r\ m(t_a) \implies \\ \qquad\qquad \exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a.\mathcal{M}(P^{S+},c,m) = t'_r\ m(t'_a) \\ \forall m, (\{m_1 \ldots m_n\}, \mathbf{e}) \in \mathcal{SC}hs(P^{S+},c,m). \\ \qquad \forall i \in \{1 \ldots n\}.Sig(P^{S+},c,m_i) = \texttt{void}\ m(t_i), \\ \qquad \mathcal{M}(P^{S+},c,m) = t\ m(t_0), \\ \qquad P^{S+}, [m_1\_\texttt{x} \mapsto t_1 \ldots m_n\_\texttt{x} \mapsto t_n, m\_\texttt{x} \mapsto t_0, \texttt{this} \mapsto c] \vdash \mathbf{e} : t \\ \forall f.\mathcal{F}(P^{S+},c,f) = c' \implies P^{S+} \vdash c'\ \diamond_{cl} \\ \forall f.\mathcal{F}(P^{S+},\mathcal{S}up(P^{S+},c),f) = t \implies \mathcal{F}(P^{S+},c,f) = t \end{array}}{P^{S+} \vdash c} \quad \text{(WFClass)}$$

$$\frac{\mathcal{S}up(P^{S+},c) \neq \mathcal{U}df \quad \vee \quad c = \texttt{Object}}{P^{S+} \vdash c\ \diamond_{cl}} \quad \text{(IsClass)}$$

Source type rules:

$$\frac{v \in \{\texttt{this}, m\_\texttt{x}\}}{P^{S+}, \Gamma \vdash v : \Gamma(v)} \quad \text{(STVar)} \qquad \frac{P^{S+} \vdash c\ \diamond_{cl}}{P^{S+}, \Gamma \vdash \texttt{null} : c} \quad \text{(STNull)}$$

$$\frac{}{P^{S+}, \Gamma \vdash \texttt{voidval} : \texttt{void}} \quad \text{(STVoid)} \qquad \frac{P^{S+} \vdash c\ \diamond_{cl}}{P^{S+}, \Gamma \vdash \texttt{new}\ c : c} \quad \text{(STNew)}$$

$$\frac{\begin{array}{c} P^{S+}, \Gamma \vdash \mathbf{e}_1 : c \\ P^{S+}, \Gamma \vdash \mathbf{e}_2 : t \\ \mathcal{M}(P^{S+},c,m) = t_r\ m(t) \end{array}}{P^{S+}, \Gamma \vdash \mathbf{e}_1.m(\mathbf{e}_2) : t_r} \quad \text{(STInv)} \qquad \frac{\begin{array}{c} P, \Gamma \vdash \mathbf{e}_1 : t_1 \\ P, \Gamma \vdash \mathbf{e}_2 : t_2 \end{array}}{P, \Gamma \vdash \mathbf{e}_1\ ;\ \mathbf{e}_2 : t_2} \quad \text{(STSeq)}$$

$$\frac{\begin{array}{c} P^{S+}, \Gamma \vdash \mathbf{e} : c \\ P^{S+}, \Gamma \vdash \mathcal{S}up(P^{S+},c) = c' \end{array}}{P^{S+}, \Gamma \vdash \mathbf{e} : c'} \quad \text{(STSub)} \qquad \frac{P^{S+}, \Gamma \vdash \mathbf{e} : \texttt{void}}{P^{S+}, \Gamma \vdash \texttt{spawn}\ \mathbf{e} : \texttt{void}} \quad \text{(STSpawn)}$$

$$\frac{\begin{array}{c} P^{S+}, \Gamma \vdash \mathbf{e} : c \\ \mathcal{F}(P^{S+},c,f) = t \end{array}}{P^{S+}, \Gamma \vdash \mathbf{e}.f : t} \quad \text{(STFld)} \qquad \frac{\begin{array}{c} P^{S+}, \Gamma \vdash \mathbf{e}_1 : c \\ \mathcal{F}(P^{S+},c,f) = t \\ P^{S+}, \Gamma \vdash \mathbf{e}_2 : t \end{array}}{P^{S+}, \Gamma \vdash \mathbf{e}_1.f := \mathbf{e}_2 : t} \quad \text{(STAss)}$$

$$\frac{}{P^{S+}, \Gamma \vdash \texttt{0} : \texttt{int}} \quad \text{(STZero)} \qquad \frac{\begin{array}{c} P^{S+}, \Gamma \vdash \mathbf{e}_1 : \texttt{int} \\ P^{S+}, \Gamma \vdash \mathbf{e}_2 : t \end{array}}{P^{S+}, \Gamma \vdash \texttt{nonzero}\ \mathbf{e}_1\ \mathbf{e}_2 : t} \quad \text{(STNonZero)}$$

$$\frac{P^{S+}, \Gamma \vdash \mathbf{e} : \texttt{int}}{P^{S+}, \Gamma \vdash \texttt{inc}\ \mathbf{e} : \texttt{int}} \quad \text{(STInc)} \qquad \frac{P^{S+}, \Gamma \vdash \mathbf{e} : \texttt{int}}{P^{S+}, \Gamma \vdash \texttt{dec}\ \mathbf{e} : \texttt{int}} \quad \text{(STDec)}$$

Figure 7.6: Rules for well-formed $\mathcal{L}^{S+}$ programs.

7.3 Definition of the translation φ_{JS+}

7.3.1 Example

Before giving the translation of general $\mathcal{L}^J$ programs into $\mathcal{L}^{S+}$ programs, we consider a fairly general example that shows how chords implement the synchronisation primitives of $\mathcal{L}^J$, i.e. mutexes and condition variables. All classes in the $\mathcal{L}^J$ program will have their method bodies converted by $\mathbb{C} : SrcExpr^J \to SrcExpr^{S+}$, and will have the same set of class members added.

```
class C {
        void f(x) {
                e
        }
}
```

This is converted into the following code:

```
class C {

        int counter;

        void lock(void lock_x) & async unlock(void unlock_x) { }

        void realWait(void realWait_x) & async realNotify(void realNotify_x) { }

        void wait(void wait_x) {
                this.counter := inc this.counter;
                this.unlock(voidval);
                this.realWait(voidval);
                this.lock(voidval);
        }

        void notify(void notify_x) {
                nonzero this.counter {
                        this.realNotify(voidval);
                        this.counter := dec this.counter;
                        voidval;
                }
        }

        void notifyAll(void notifyAll_x) {
                nonzero this.counter {
                        this.notify(voidval);
                        this.notifyAll(voidval);
                }
        }

        void f(m_f) {
                C(e)[m_f/x];
        }
}
```

Of the added members, we consider the chord of `realWait()` and `realNotify()`, and the field `counter`, to be "private" to the translation, because they are not directly accessed by any of the code that was maintained from the $\mathcal{L}^J$ program, in this case $\mathbb{C}(e)$.

The new functions `lock()`, `unlock()`, `wait()`, `notify()`, and `notifyAll()` are meant to have the same functionality as their counterparts in $\mathcal{L}^J$, although here they are not language constructs, they are methods whose bodies use the "private" members to synchronise and hold state.

The translation $\mathbb{C} : SrcExpr^J \to SrcExpr^{S+}$ converts the `lock` e, `unlock` e, `wait` e, `notify` e, and `notifyAll` e to the appropriate method calls (e.g. e.`lock` (`voidval`)). It is homomorphic over the rest of the constructs in $\mathcal{L}^J$, since they are common to both languages.

We now study the implementation of the new methods. Firstly, the behaviour of the synchronous chord that is comprised of `lock()` and `unlock()` should be familiar to us. This is the standard way to implement a mutex with chords, and we have used it many times already in this report. There is never more than one message on the `unlock()` queue, which is empty if and only if there is a thread in the critical section defined by the mutex.

The interesting part of this translation is not the mutexes, but how to implement the condition variables in $\mathcal{L}^J$, i.e. `wait` e, `notify` e and `notifyAll` e. We recall that the $\mathcal{L}^J$ semantics do not define progress unless the thread that executes any of these constructs has the lock on the associated object. Correct programs do not encounter this situation at run-time.

After translation of all source code by $\mathbb{C}$, these constructs are replaced with the method calls in question, so we know that these methods are only called by threads with the lock. only one thread can be executing any of these methods (on a specific object) at any one time, with the exception that another thread can be part-way through the execution of `wait()`. This means we do not have to worry about the fact that `counter` is a shared memory variable, as it is protected by the mutex of the class.

The field variable `counter` is of `int` type, and keeps count on the number of threads waiting for notification. It is used so that `notify()` can avoid emitting the `realNotify()` message when there are no threads to wake up (so no notifications are queued), and also so that `notifyAll()` can emit the right number of messages to wake up all the sleeping threads.

The function `wait()` is quite simple. Its behaviour is the same regardless of the state of the object, it simply increments the counter, gives up its lock of the object, and waits for a `realNotify()` message to occur. When this happens, it re-acquires the lock (as the semantics of $\mathcal{L}^J$ require) and returns.

The function `notify()` does nothing if no threads are currently waiting, i.e. if `counter` is zero. If `counter` is non-zero, there are that many threads blocked in a call to the synchronous method `realWait()`, i.e. they are unable to progress from their run-time expression of $E[\iota$.`realWait` (`voidval`)]. So if `counter` is non-zero, then a single `realNotify()` message is emitted, and the function returns. The `realNotify()` message is queued, and can either join immediately with one of the a threads that is blocked, or join at some later stage, after some unrelated steps of execution in threads that are not waiting. The moment that the join occurs is the moment that the wakeup happens, all the steps leading up to this event are preparation. (The `voidval` in the function is there so that the body

of the conditional types to `void`, otherwise the code is not well-typed.)

The function `notifyAll()` also does nothing if no threads are currently waiting, i.e. if `counter` is zero. It uses recursion to repeatedly execute `notify()` until `counter` is zero. I.e., it calls `notify()` once for every thread that was waiting, thus notifying them all, and filling the `realNotify()` queue with enough messages for them all to join. No threads can enter a waiting state while this is happening because of the mutex that must be held when `notifyAll()` is called.

7.3.2 Translation

The encoding is given in figure 7.7. Because the encoding can translate into a range of equivalent programs, since we can choose almost any name we like for the new class members, we collect this flexibility into a single "decision" function δ and specify this as an argument to φ_{JS+}.

$$\delta : \Delta = Id_m \times Id_m \times Id_m \times Id_m \times Id_m \times Id_m \times Id_m \times Id_f$$

For all classes, $\delta = (lock, unlock, wait, notify, notifyAll, realNotify, realWait, init, counter)$ tells us the names of all the methods added to the class to implement the semantics of $\mathcal{L}^J$'s mutexes and condition variables. In previous translations, the names of new methods have been chosen for each class specifically, but now we use the same name for all classes. Since the methods do the same thing and have the same type signatures across all the classes, we do not have any clashes between classes due to inheritance.

The result of $\varphi_{JS+}(P^J, \delta)$ is only well-defined when δ is well-formed, i.e. that $P^J \vdash \delta$ as defined below:

$$\begin{array}{lll} P^S \vdash \delta & \iff & \delta = (lock, unlock, wait, notify, notifyAll, realNotify, realWait, init, counter) \\ & & \forall c \;.\; \mathcal{F}(P^J, c, counter) = \mathcal{U}df \\ & & \text{let } S = \{lock, unlock, wait, notify, notifyAll, realNotify, realWait, init\} \\ & & |S| = 8 \qquad \text{(all are distinct)} \\ & & \forall c,\; m \in S \;.\; \mathcal{M}(P^J, c, m) = \mathcal{U}df \end{array}$$

This ensures that *counter* does not conflict with any existing fields, and that all the new method names are distinct, and none conflict with any existing methods in any of the classes in the original program.

7.3.3 Properties of the translation

It can be immediately seen from the definition, that if $P^{S+} = \varphi_{JS+}(P^J, \delta)$:

- $\forall c, m. |\mathcal{SCHs}(P^{S+}, c, m)| = 1$ We have no use for this kind of non-determinism when implementing monitors with chords.
- $\forall c. \mathcal{Q}(P^{S+}, c) = \{\delta_2, \delta_7\}$ The only queues we use are for implementing the behaviour of a mutex and the blocking and releasing of threads.

$\varphi_{JS+}(P^J, \delta) = P^{S+}$ if and only if

$$\mathcal{SChs}(P^{S+}, c, m) = \begin{cases}
\{(\emptyset, \mathbb{C}(\mathbf{e})[m_\mathtt{x}/\mathtt{x}])\} & \text{if } \mathcal{Meth}(P^J, c, m) = \mathbf{e} \\
\{(\{\delta\downarrow_2\}, \mathtt{voidval})\} & \text{if } m = \delta\downarrow_1 \\
\{(\{\delta\downarrow_7\}, \mathtt{voidval})\} & \text{if } m = \delta\downarrow_6 \\
\{\ (\emptyset, \mathtt{this}.\delta\downarrow_9 := \mathtt{inc}\ \mathtt{this}.\delta\downarrow_9; & \\
\quad \mathtt{this}.\delta\downarrow_2(\mathtt{voidval}); & \\
\quad \mathtt{this}.\delta\downarrow_6(\mathtt{voidval}); & \\
\quad \mathtt{this}.\delta\downarrow_1(\mathtt{voidval}))\ \} & \text{if } m = \delta\downarrow_3 \\
\{\ (\emptyset, \mathtt{nonzero}\ \mathtt{this}.\delta\downarrow_9\ (& \\
\quad \mathtt{this}.\delta\downarrow_7(\mathtt{voidval}); & \\
\quad \mathtt{this}.\delta\downarrow_9 := \mathtt{dec}\ \mathtt{this}.\delta\downarrow_9; & \\
\quad \mathtt{voidval} & \\
\quad))\ \} & \text{if } m = \delta\downarrow_4 \\
\{\ (\emptyset, \mathtt{nonzero}\ \mathtt{this}.\delta\downarrow_9\ (& \\
\quad \mathtt{this}.\delta\downarrow_4(\mathtt{voidval}); & \\
\quad \mathtt{this}.m(\mathtt{voidval}) & \\
\quad))\ \} & \text{if } m = \delta\downarrow_5 \\
\{(\emptyset, \mathtt{this}.\delta\downarrow_2(\mathtt{voidval}); \mathtt{this})\} & \text{if } m = \delta\downarrow_8
\end{cases}$$

$$\mathcal{M}(P^{S+}, c, m) = \begin{cases}
\mathcal{M}(P^J, c, m) & \text{if } \mathcal{M}(P^J, c, m) \neq \mathcal{U}df \\
\mathtt{void}\ m(\mathtt{void}) & \text{if } \exists i \in \{1 \ldots 7\}.m = \delta\downarrow_i \\
c\ m(\mathtt{void}) & \text{if } m = \delta\downarrow_8
\end{cases}$$

$$\mathcal{F}(P^{S+}, c, f) = \begin{cases}
\mathcal{F}(P^J, c, f) & \text{if } \mathcal{F}(P^J, c, f) \neq \mathcal{U}df \\
\mathtt{int} & \text{if } f = \delta\downarrow_9
\end{cases}$$

$\mathcal{Sup}(P^{S+}, c) = \mathcal{Sup}(P^J, c)$

$$\begin{array}{lcl}
\mathbb{C}(\mathtt{null}) & = & \mathtt{null} \\
\mathbb{C}(\mathtt{voidval}) & = & \mathtt{voidval} \\
\mathbb{C}(\mathtt{this}) & = & \mathtt{this} \\
\mathbb{C}(x) & = & x \\
\mathbb{C}(\mathtt{new}\ c) & = & \mathtt{new}\ c.\delta\downarrow_8(\mathtt{voidval}) \\
\mathbb{C}(\mathbf{e}_1.m(\mathbf{e}_2)) & = & \mathbb{C}(\mathbf{e}_1).m(\mathbb{C}(\mathbf{e}_2)) \\
\mathbb{C}(\mathbf{e}_1; \mathbf{e}_2) & = & \mathbb{C}(\mathbf{e}_1); \mathbb{C}(\mathbf{e}_2) \\
\mathbb{C}(\mathtt{spawn}\ \mathbf{e}) & = & \mathtt{spawn}\ \mathbb{C}(\mathbf{e}) \\
\mathbb{C}(\mathbf{e}.f) & = & \mathbb{C}(\mathbf{e}).f \\
\mathbb{C}(\mathbf{e}_1.f := \mathbf{e}_2) & = & \mathbb{C}(\mathbf{e}_1).f := \mathbb{C}(\mathbf{e}_2) \\
\mathbb{C}(\mathtt{lock}\ \mathbf{e}) & = & \mathbb{C}(\mathbf{e}).\delta\downarrow_1(\mathtt{voidval}) \\
\mathbb{C}(\mathtt{unlock}\ \mathbf{e}) & = & \mathbb{C}(\mathbf{e}).\delta\downarrow_2(\mathtt{voidval}) \\
\mathbb{C}(\mathtt{wait}\ \mathbf{e}) & = & \mathbb{C}(\mathbf{e}).\delta\downarrow_3(\mathtt{voidval}) \\
\mathbb{C}(\mathtt{notify}\ \mathbf{e}) & = & \mathbb{C}(\mathbf{e}).\delta\downarrow_4(\mathtt{voidval}) \\
\mathbb{C}(\mathtt{notifyAll}\ \mathbf{e}) & = & \mathbb{C}(\mathbf{e}).\delta\downarrow_5(\mathtt{voidval})
\end{array}$$

Figure 7.7: Translation $\varphi_{JS+} : (\mathcal{L}^J \times \Delta) \to \mathcal{L}^{S+}$

7.4 Preservation of structure (proof)

We first formally define the property of structure preservation for our programs P^J and P^{S+} where $P^{S+} = \varphi_{JS+}(P^J, \delta)$. We refer to section 3.1.2, where the contribution of [5] is put into the context of our object-oriented formalisations. In order for the translation to be structure-preserving, it must satisfy:

- The only constructs in the source expressions of $\mathcal{L}^J$ that are not present in $\mathcal{L}^{S+}$ are the five constructs that implement mutual exclusion. $\mathbb{C}$ must be homomorphic over all the other constructs. This means that the method signatures must remain intact as otherwise it would not be possible to prove well-formedness of P^S since $\mathbb{C}$ must be homomorphic over method calls.

  We can see from the definition of $\mathbb{C}$ that the only constructs over which $\mathbb{C}$ is not homomorphic are `new` c and the five constructs which represent the monitors. For `new` c we have special exception, as justified in section 3.1.4. The result of $\mathbb{C}$ when applied to the five monitor constructs is a simple macro translation where the sub-expression is used exactly once.

- Method signatures are common to $\mathcal{L}^J$ and $\mathcal{L}^{S+}$, so they must not be altered, although we do allow the addition of new methods.
  $\forall c, m.\mathcal{M}(P^J, c, m) = msig \Rightarrow \mathcal{M}(P^{S+}, c, m) = msig$

  From $P^J \vdash \delta$ we know that the three cases for the definition of $\mathcal{M}(P^{S+}, c, m)$ in φ_{JS+} are exclusive, so when $\mathcal{M}(P^J, c, m) = msig$ i.e. $\mathcal{M}(P^J, c, m) \neq \mathcal{U}df$, we know that $\mathcal{M}(P^{S+}, c, m) = \mathcal{M}(P^J, c, m) = msig$. □

- Methods are present in $\mathcal{L}^J$ but not in $\mathcal{L}^{S+}$, so the translation must replace them with some other code, while preserving the method bodies aside from calling $\mathbb{C}$ on them.

  The first case of $\mathcal{SC}hs(P^{S+}, c, m)$ in φ_{JS+} forms a synchronous chord with $\mathbb{C}(e)$ as a body, so $\forall c, m, e.\mathcal{M}eth(P^J, c, m) = e \Longrightarrow (\emptyset, \mathbb{C}(e)[m\_\mathtt{x}/\mathtt{x}]) \in \mathcal{SC}hs(P^{S+}, c, m)$, which satisfies the condition considering the specific exemption of argument renaming from the requirements, as justified in section 3.1.4. □

- Fields are a feature of $\mathcal{L}^J$ and $\mathcal{L}^{S+}$ so like method signatures, we must show they are preserved over the translation. $\forall c, f.\mathcal{F}(P^J, c, f) = c' \Rightarrow \mathcal{F}(P^{S+}, c, f) = c'$

  From $P^J \vdash \delta$ we know that the two cases for the definition of $\mathcal{F}(P^{S+}, c, f)$ in φ_{JS+} are exclusive, so when $\mathcal{F}(P^J, c, f) = c'$ i.e. $\mathcal{F}(P^J, c, f) \neq \mathcal{U}df$, we know that $\mathcal{F}(P^{S+}, c, f) = \mathcal{F}(P^J, c, f) = c'$. □

- The feature of classes is common to $\mathcal{L}^J$ and $\mathcal{L}^{S+}$, so the structure of classes must be preserved, although we allow the addition of new classes:
  $\forall c.\mathcal{S}up(P^J, c) = c' \Longrightarrow \mathcal{S}up(P^{S+}, c) = c'$

  We can prove this by inspection of φ_{JS+}, which actually has the stronger property that no classes are added or removed. □

7.5 Preservation of well-formedness (proof)

We need to establish that the well-formedness of a program is preserved when the program is translated, this is the notion of preservation of programness in [5]. Let $P^{S+} = \varphi_{JS+}(P^J, \delta)$. We must show that $\vdash P^J \Longrightarrow \vdash P^{S+}$.

Lemma 7.5.1 *If $P^{S+} = \varphi_{JS+}(P^J, \delta)$ then $P^J \vdash c \diamond_{cl} \Longrightarrow P^{S+} \vdash c \diamond_{cl}$*
Proof:

$$\begin{array}{lll} P^J \vdash c \diamond_{cl} & \Longrightarrow & \mathcal{S}up(P^J, c) \neq \mathcal{U}df \vee c = \texttt{Object} \quad \text{(IsClass)} \\ & \Longrightarrow & \mathcal{S}up(P^{S+}, c) \neq \mathcal{U}df \vee c = \texttt{Object} \quad \text{(Def } \varphi_{JS+}\text{)} \\ & \Longrightarrow & P^{S+} \vdash c \diamond_{cl} \quad \text{(IsClass)} \end{array}$$

Lemma 7.5.2 *Method type signatures are preserved over inheritance in the translated program: If $\vdash P^J$ and $P^{S+} = \varphi_{JS+}(P^J, \delta)$ then*
$\forall m. \mathcal{M}(P^{S+}, \mathcal{S}up(P^{S+}, c), m) = t_r\ m(t_a) \Longrightarrow \exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a\ .\ \mathcal{M}(P^{S+}, c, m) = t'_r\ m(t'_a)$.
Proof:

Let $\vdash P^J$ and $\mathcal{M}(P^{S+}, \mathcal{S}up(P^{S+}, c), m) = t_r\ m(t_a)$. We know from the definition of φ_{JS+} that there are three possibilities:

- $\mathcal{M}(P^J, \mathcal{S}up(P^J, c), m) = t_r\ m(t_a) \neq \mathcal{U}df$ in which case from $P^J \vdash c$ we know that $\exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a . \mathcal{M}(P^J, c, m) = t'_r\ m(t'_a)$, and from the definition of φ_{JS+} we know that $\mathcal{M}(P^{S+}, c, m) = t'_r\ m(t'_a)$ □
- $\exists i \in \{1 \ldots 7\}. m = \delta{\downarrow_i}$, in which case $t_r = t_a = \texttt{voidval}$, and also $\mathcal{M}(P^{S+}, c, m) = \texttt{voidval}\ m(\texttt{voidval})$ because $\delta{\downarrow_i}$ is independent of class. Clearly $\texttt{voidval} \sqsubseteq \texttt{voidval}$ and $\texttt{voidval} \sqsupseteq \texttt{voidval}$. □
- $m = \delta{\downarrow_8}$ in which case $t_r = \mathcal{S}up(P^{S+}, c)$ and $t_a = \texttt{void}$. We also know that $\mathcal{M}(P^{S+}, c, m) = c\ m(\texttt{voidval})$ since $\delta{\downarrow_8}$ is independent of class. Clearly $\mathcal{S}up(P^{S+}, c) \sqsubseteq c$ and $\texttt{voidval} \sqsupseteq \texttt{voidval}$. □

Lemma 7.5.3 *Well-typedness of expressions is preserved over the translation:*
If $P^{S+} = \varphi_{JS+}(P^J, \delta)$,
$(m_1 \ldots m_n, \boldsymbol{e}^S_{ch}) = \mathcal{M}eth(P^S, c, m)$,
$\mathcal{M}(P^S, c, m) = \_\ m(t_0)$, $\mathcal{M}(P^S, c, m_i) = \_\ m_i(t_i)$ (for all $i \in \{1 \ldots n\}$),
$\Gamma = [m_1\_\boldsymbol{x} \mapsto t_1 \ldots m_n\_\boldsymbol{x} \mapsto t_n, m\_\boldsymbol{x} \mapsto t_0, \boldsymbol{this} \mapsto c]$, and
$\boldsymbol{e}^S$ is a subterm of $\boldsymbol{e}^S_{ch}$, then
$P^J, \Gamma \vdash \boldsymbol{e}^J : t \Longrightarrow P^{S+}, \Gamma \vdash \mathbb{C}(\boldsymbol{e}^J) : t$
Proof: Induction over the structure of derivations.

- STVar, STVoid: Trivial. □

- STNull, use lemma 7.5.1. □
- STNew: Note that $\mathbb{C}(\texttt{new } c) = \texttt{new } c.\delta\downarrow_8(\texttt{voidval})$ and that $\mathcal{M}(P^{S+}, c, \downarrow_8) = c\ m(\texttt{void})$, using STNew and STInv, $P^{S+}, \Gamma \vdash \mathbb{C}(\texttt{new } c) : c$. □
- STInv: From the definition of φ_{JS+},
  $\mathcal{M}(P^J, c, m) = t\ m(t_a) \neq \mathcal{U}df \implies \mathcal{M}(P^{S+}, c, m) = t\ m(t_a)$. Using this and the induction hypothesis we can show $P^{S+}, \Gamma \vdash \mathbb{C}(\mathbf{e}_1.m(\mathbf{e}_2)) : t$. □
- STSeq: Use that $\mathbb{C}(\mathbf{e}_1; \mathbf{e}_2) = \mathbb{C}(\mathbf{e}_1); \mathbb{C}(\mathbf{e}_2)$, and the induction hypothesis. □
- STSub: Use $\mathcal{S}up(P^{S+}, c) = \mathcal{S}up(P^J, c)$ from definition of φ_{JS+}, and the induction hypothesis. □
- STSpawn: Using that $\mathbb{C}(\texttt{spawn } \mathbf{e}) = \texttt{spawn } \mathbb{C}(\mathbf{e})$ and the induction hypothesis, trivial. □
- STWait: Using that $\mathbb{C}(\texttt{wait } \mathbf{e}) = \mathbb{C}(\mathbf{e}).\delta\downarrow_3(\texttt{voidval})$ and $\forall c.\mathcal{M}(P^{S+}, c, \delta\downarrow_3) = \texttt{void}\ \delta\downarrow_3(\texttt{void})$, we know that $P^{S+}, \Gamma \vdash \mathbb{C}(\texttt{wait } \mathbf{e}) : \texttt{void}$. □
- STLock, STUnlock, STNotify, STNotifyAll: Same as STWait. □

Lemma 7.5.4 *If $P, \Gamma \vdash \boldsymbol{e} : t$ and $\Gamma'(v[m\_\boldsymbol{x}/\boldsymbol{x}]) = \Gamma(v)$ for $v \in \{\boldsymbol{x}, \boldsymbol{this}\}$, then $P, \Gamma' \vdash \boldsymbol{e}[m\_\boldsymbol{x}/\boldsymbol{x}] : t$*

Proof: Induction over the structure of derivations.

- Only interesting case is (STVar). We know from the the rule $\mathbf{e} = v \in \{\texttt{x}, \texttt{this}\}$, and from the premise of the lemma $\Gamma'(v[m\_\texttt{x}/\texttt{x}]) = \Gamma(v) = t$ so $P, \Gamma' \vdash v[m\_\texttt{x}/\texttt{x}] : t$. □

Lemma 7.5.5 $P^J \vdash c \implies P^{S+} \vdash c$

Proof:

$$\begin{array}{lll} P^J \vdash c & \implies & P^J \vdash \mathcal{S}up(P^J, c)\ \diamond_{cl} \qquad \text{(WFClass)} \\ & \implies & P^J \vdash \mathcal{S}up(P^{S+}, c)\ \diamond_{cl} \qquad \text{(Def } \varphi_{JS+}\text{)} \\ & \implies & P^{S+} \vdash \mathcal{S}up(P^{S+}, c)\ \diamond_{cl} \qquad \text{(Lemma 7.5.1)} \end{array}$$

$$\begin{array}{lll} P^J \vdash c & \implies & \forall m.\mathcal{M}(P^{S+}, \mathcal{S}up(P^{S+}, c), m) = t_r\ m(t_a) \\ & & \Rightarrow \exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a\ .\ \mathcal{M}(P^{S+}, c, m) = t'_r\ m(t'_a) \qquad \text{(Lemma 7.5.2)} \end{array}$$

For all methods m and synchronous chords $(\{m_1 \ldots m_n\}, \mathbf{e}^{S+}) \in \mathcal{SC}hs(P^{S+}, c, m)$ either

- $\mathcal{M}eth(P^J, c, m) = \mathbf{e}^J \neq \mathcal{U}df$ where $\mathbf{e}^{S+} = \mathbb{C}(\mathbf{e}^J)[m\_\texttt{x}/\texttt{x}]$, in which case:
  - The value of $n = 0$

- $\mathcal{M}(P^J, c, m) = t_r\ m(t_a)$ $(P^J \vdash c)$ and so $\mathcal{M}(P^{S+}, c, m) = t_r\ m(t_a)$ by the definition of φ_{JS+}.
- $P^J, \Gamma \vdash \mathbf{e}^J : t_r$ (where $\Gamma = [\mathtt{x} \mapsto t_a, \mathtt{this} \mapsto c]$) (WFClass) and so $P^{S+}, \Gamma' \vdash \mathbf{e}^{S+} : t_r$ where $\Gamma' = [m\_\mathtt{x} \mapsto t_a, \mathtt{this} \mapsto c]$ (using lemma 7.5.2 and lemma 7.5.4). □

- $n = 1$, $m = \delta\downarrow_1$, $m_1 = \delta\downarrow_2$, (also applies for the other case where $m = \delta\downarrow_6$, $m_1 = \delta\downarrow_7$) and $\mathbf{e}^{S+} = \mathtt{voidval}$, in which case:
  - By the definition of φ_{JS+}, we know that $\mathcal{M}(P^{S+}, c, m_1) = \mathtt{void}\ m_1(\mathtt{void})$ and also $\mathcal{M}(P^{S+}, c, m) = \mathtt{void}\ m(\mathtt{void})$.
  - Using STVoid, $P^{S+}, \Gamma \vdash \mathbf{e}^A : \mathtt{void}$ (where $\Gamma = [m_1\_\mathtt{x} \mapsto \mathtt{void}, m\_\mathtt{x} \mapsto \mathtt{void}, \mathtt{this} \mapsto c]$). □

- $n = 0$, $m = \delta\downarrow_3$, and $\mathbf{e}^{S+} = \mathtt{this}.\delta\downarrow_9 := \mathtt{inc}\ \mathtt{this}.\delta\downarrow_9$; $\mathtt{this}.\delta\downarrow_2(\mathtt{voidval}); \mathtt{this}.\delta\downarrow_6(\mathtt{voidval}); \mathtt{this}.\delta\downarrow_1(\mathtt{voidval})$, in which case:
  - By the definition of φ_{JS+}, we know that $\mathcal{M}(P^{S+}, c, m) = \mathtt{void}\ m(\mathtt{void})$, and $\forall i \in \{2, 6, 1\}.\mathcal{M}(P^{S+}, c, \delta\downarrow_i) = \mathtt{void}\ \delta\downarrow_i(\mathtt{void})$. We also know that $\mathcal{F}(P^{S+}, c, \delta\downarrow_9) = \mathtt{int}$.
  - Let $\Gamma = [m\_\mathtt{x} \mapsto \mathtt{void}, \mathtt{this} \mapsto c]$. Using (STFld) and (STVar), $P^{S+}, \Gamma \vdash \mathtt{this}.\delta\downarrow_9 : \mathtt{int}$. This means that $P^{S+}, \Gamma \vdash \mathtt{inc}\ \mathtt{this}.\delta\downarrow_9 : \mathtt{int}$, and thus $P^{S+}, \Gamma \vdash \mathtt{this}.\delta\downarrow_9 := \mathtt{inc}\ \mathtt{this}.\delta\downarrow_9 : \mathtt{int}$. It is trivial to type all the method calls to `void`, and therefore the composition can also be typed: $P^{S+}, \Gamma \vdash \mathbf{e}^{S+} : \mathtt{void}$ □

- $n = 0$, $m = \delta\downarrow_4$, and $\mathbf{e}^{S+} = \mathtt{nonzero}\ \mathtt{this}.\delta\downarrow_9\ (\mathtt{this}.\delta\downarrow_7(\mathtt{voidval}); \mathtt{this}.\delta\downarrow_9 := \mathtt{dec}\ \mathtt{this}.\delta\downarrow_9; \mathtt{voidval})$, in which case:
  - By the definition of φ_{JS+}, we know that $\mathcal{M}(P^{S+}, c, m) = \mathtt{void}\ m(\mathtt{void})$, and $\mathcal{M}(P^{S+}, c, \delta\downarrow_7) = \mathtt{void}\ \delta\downarrow_7(\mathtt{void})$. We also know that $\mathcal{F}(P^{S+}, c, \delta\downarrow_9) = \mathtt{int}$.
  - Let $\Gamma = [m\_\mathtt{x} \mapsto \mathtt{void}, \mathtt{this} \mapsto c]$. Using (STFld) and (STVar), $P^{S+}, \Gamma \vdash \mathtt{this}.\delta\downarrow_9 : \mathtt{int}$. This means that $P^{S+}, \Gamma \vdash \mathtt{dec}\ \mathtt{this}.\delta\downarrow_9 : \mathtt{int}$, and thus $P^{S+}, \Gamma \vdash \mathtt{this}.\delta\downarrow_9 := \mathtt{dec}\ \mathtt{this}.\delta\downarrow_9 : \mathtt{int}$. It is trivial to type the method call to `void`, and the `voidval` types to `void`, so the composition of these three statements is typed to `void`. We can therefore type the whole body: $P^{S+}, \Gamma \vdash \mathbf{e}^{S+} : \mathtt{void}$ □

- $n = 0$, $m = \delta\downarrow_5$, and $\mathbf{e}^{S+} = \mathtt{nonzero}\ \mathtt{this}.\delta\downarrow_9\ (\mathtt{this}.\delta\downarrow_4(\mathtt{voidval}); \mathtt{this}.m(\mathtt{voidval}))$, in which case:
  - By the definition of φ_{JS+}, we know that $\mathcal{M}(P^{S+}, c, m) = \mathtt{void}\ m(\mathtt{void})$, and $\mathcal{M}(P^{S+}, c, \delta\downarrow_4) = \mathtt{void}\ \delta\downarrow_4(\mathtt{void})$. We also know that $\mathcal{F}(P^{S+}, c, \delta\downarrow_9) = \mathtt{int}$.

  - Let $\Gamma = [m\_\texttt{x} \mapsto \texttt{void}, \texttt{this} \mapsto c]$. Using (STFld) and (STVar), $P^{S+}, \Gamma \vdash \texttt{this}.\delta\downarrow_9 : \texttt{int}$. We now type the second sub-expression of the `nonzero` block. It is trivial to type the method calls to `void`, and thus the composition of the two statements is typed to `void`. We can therefore type the whole body: $P^{S+}, \Gamma \vdash \mathbf{e}^{S+} : \texttt{void}$ □

- $m = \delta\downarrow_8$, $n = 0$ and $\mathbf{e}^{S+} = \texttt{this}.\delta\downarrow_2(\texttt{voidval}); \texttt{this}$
  - We know from the definition of φ_{JS+} that $\mathcal{M}(P^{S+}, c, m) = c\ m(\texttt{void})$.
  - We also know that $\mathcal{M}(P^{S+}, c, \delta\downarrow_2) = \texttt{void}\ \delta\downarrow_2(\texttt{void})$.
  - We can now show that $P^{S+}, [m\_\texttt{x} \mapsto \texttt{void}, \texttt{this} \mapsto c] \vdash \mathbf{e}^{S+} : c$ because the call to $\texttt{this}.\delta\downarrow_2(\texttt{voidval})$ can be typed to `void`, and `this` is trivially typed to c, thus the composition can be typed to c.

For all fields f such that $\mathcal{F}(P^{S+}, c, f) = c'$, we know from the definition of φ_{JS+} that $\mathcal{F}(P^J, c, f) = c'$. From $P^J \vdash c$ we therefore know that $P^J \vdash c' \diamond_{cl}$ and from lemma 7.5.1 we know that $P^{S+} \vdash c' \diamond_{cl}$.

For all fields f such that $\mathcal{F}(P^{S+}, \mathcal{S}up(P^{S+}, c), f) = t$ we know from the definition of φ_{JS+} that there are two possibilities:

- $\mathcal{F}(P^J, \mathcal{S}up(P^{S+}, c), f) = t$ and thus $\mathcal{F}(P^J, \mathcal{S}up(P^J, c), f) = t$, in which case we know from $P^J \vdash \mathcal{S}up(P^J, c)$ that $\mathcal{F}(P^J, c) = t$ and thus by the definition of φ_{JS+}: $\mathcal{F}(P^{S+}, c, f) = t$. □
- $f = \delta\downarrow_9$ in which case $t = \texttt{int}$, and because δ is independent of class, $\mathcal{F}(P^{S+}, c, f) = \texttt{int}$ as well. □

The above is sufficient to prove $P^{S+} \vdash c$. □

Theorem 7.5.6 $\vdash P^J \Longrightarrow \vdash P^{S+}$.

Proof:

$$
\begin{array}{rcll}
\vdash P^J & \Longrightarrow & \forall c.\mathcal{S}up(P^J, c) \neq \mathcal{U}df \Rightarrow P^J \vdash c & \text{(WFProg)} \\
& \Longrightarrow & \forall c.\mathcal{S}up(P^{S+}, c) \neq \mathcal{U}df \Rightarrow P^J \vdash c & \text{(Def } \varphi_{JS+}\text{)} \\
& \Longrightarrow & \forall c.\mathcal{S}up(P^{S+}, c) \neq \mathcal{U}df \Rightarrow P^{S+} \vdash c & \text{(Lemma 7.5.5)} \\
& \Longrightarrow & \vdash P^{S+} & \text{(WFProg)}
\end{array}
$$

Chapter 8

$\mathcal{L}^S$ is at most as expressive as $\mathcal{L}^{J*}$

This chapter concerns the relation between chords and monitors, but in the opposite direction to chapter 7, i.e. we implement the synchronisation of chords with monitors. We define a translation from the language $\mathcal{L}^S$ into a new language $\mathcal{L}^{J*}$.

The language $\mathcal{L}^{J*}$ is a version of $\mathcal{L}^J$ with features that define special queue fields and operations to act on them. Java does not have these features, but can encode them as will be shown below. Thus they would add nothing to the expressiveness of Java. The presence of queues in the formalisation means $\mathcal{L}^{J*}$ does not need fields, as demonstrated in [4]. Thus $\mathcal{L}^{J*}$ is not strictly an extension of $\mathcal{L}^J$ like $\mathcal{L}^{S+}$ was of $\mathcal{L}^S$. Another way of looking at this is – because we are translating from a language that does not have fields, and the translation does not use fields for its own mechanisms, we do not need fields in $\mathcal{L}^{J*}$.

It does not seem to be possible to show that $\mathcal{L}^{J*}$ and $\mathcal{L}^J$ are equivalent, since to encode the queues in $\mathcal{L}^J$ requires some way of specifying non-determinism in programs which is not present in $\mathcal{L}^J$ (but is present in Java because Java has a pseudo-random number generator). There is non-determinism in $\mathcal{L}^J$ in the way that threads are interleaved, but it does not seem likely that this is "powerful enough" to express the non-determinism of queue selection in an elegant way.

8.1 Definition of $\mathcal{L}^{J*}$

We formalise $\mathcal{L}^{J*}$, an object-oriented language with monitors, like $\mathcal{L}^J$ but without fields, and with features added to represent a "queue" library. We motivate and explain the design of the formalism, and give a Java implementation of the features that model the "queue" library.

8.1.1 Method arguments

The encoding to $\mathcal{L}^{J*}$ needs multiple arguments to method calls. There is no obvious encoding of multiple arguments (within a concurrent formalisation) when we have conventional methods as opposed to chords. In the sequential world, we can use fields as a substitute for multiple argument passing, but since fields are a shared state in the concurrent world, this is no longer feasible since different threads will interfere with each other.

In $\mathcal{L}^S$ and $\mathcal{L}^A$, a chord body can access a set of arguments, since every chord has a set of associated methods, and each method has its own argument. These arguments were named after their respective methods, and identified with the lexicon $m\_\texttt{x}$. Since we now have a single method per method body, but each method now can have multiple arguments, we consider the arguments to be an ordered sequence, and elements of this sequence are referenced with an index number. The sequence of arguments is therefore $x_1 \ldots x_n$.

8.1.2 Features in $\mathcal{L}^{J*}$ and not in real languages

The semantics of chords includes queues and various operations on queues, some of which are have non-deterministic semantics. Unfortunately such things cannot be trivially represented with the abstract formalisations in which we work (because of the lack of many language features that are designed for defining such things), so we augment the formalisation with language features to do precisely what is required to implement chords.

We justify this by claiming that a further translation into a more realistic language could be made by implementing these artificial features with conventional code. This claim will be backed up later with the code listing of such an implementation.

The essential issue here is not how queues can be implemented, but how the synchronisation behaviour can be uncoupled from the method invocations and implemented with method bodies and locks. The decision to encode the queue as an abstract data type within the language itself, should not be seen as a failure or omission, but as a technique for focussing on the important issues and keeping the encoding as simple as possible. The features added to represent a queue implementation are as follows:

- In the definition of programs, it is specified for each class, what queues will be available in that class. The queues are identified using a method name, e.g. m. Any queue m in a class must have a corresponding method m, the type of element that exists in that queue is the same as the type of the argument of that method.

- In the definition of an object on the heap, there is a structure for holding a multiset of values for each queue. This is identical to the representation of queues in $\mathcal{L}^A$ and $\mathcal{L}^S$.

- In source and run-time expressions, there are three extra constructs for operating on the queues in each object:

  - `push` x `this` m is defined to add the value of the argument x to queue m of `this`. This emulates the queue behaviour used during asynchronous method call. We could have defined arbitrary expressions e_1, e_2 instead of using x and `this`. However we feel that since we only need these specific atoms to do the translation, and if we were to allow arbitrary expressions then we would need a larger definition of contexts to define the reduction of these expressions before the execution of `push` $v\ \iota\ m$.
  - `pop this` m is defined to remove an arbitrary (so this is a source of non-determinism) element from the queue m in `this`. The expression reduces to the value taken from the queue.

– `ndcond this` $(m_1^1 \dots m_{n^1}^1, \mathbf{e}^1) \dots (m_1^k \dots m_{n^k}^k, \mathbf{e}^k)(\mathbf{e})$ is much more complicated. It is an expression which reduces to one of the $\mathbf{e}^1 \dots \mathbf{e}^k$ or $\mathbf{e}$ sub-expressions within it, depending on the state of the queues in the object `this`. Each pair $(m_1^i \dots m_{n^i}^i, \mathbf{e}^i)$ contains a set of queues and an expression. The pair is "suitable" if none of the referenced queues are empty. If no pairs are suitable, the expression evaluates to the "default" sub-expression $\mathbf{e}$. Otherwise, the expression evaluates to the sub-expression in one of the suitable pairs, the choice of which pair, when more than one is suitable, is arbitrary (another source of non-determinism). This is meant to emulate the semantics of chord body invocation. The token `ndcond` stands for "non-deterministic condition".

There is additionally non-determinism in $\mathcal{L}^S$ (and $\mathcal{L}^A$) if a method is both a synchronous part of a chord, and an asynchronous part of a chord. Invocation of such a method can either choose the synchronous invocation rule if the state of the queues is appropriate, or the asynchronous rule and append its argument to the associated queue. When representing this in $\mathcal{L}^{J*}$ we have trouble defining a method that can have two possible behaviours. To remedy this, we unrealistically specify a set of method bodies for each method, and allow one to be arbitrarily chosen during invocation. Such a feature can be implemented in conventional Java using an `if` statement and the pseudo-random number generator, so the addition of this unrealistic feature does not affect our results.

8.1.3 Implementing the queue operations in Java

By giving a structure preserving translation from the $\mathcal{L}^{J*}$ constructs that model queue operations to Java, we show that $\mathcal{L}^{J*}$ is no more expressive than Java. We can consider two possible implementations of the $\mathcal{L}^{J*}$ constructs in question – either a "fair" implementation where all the possible behaviours occur equally frequently, or an "unfair" implementation where this is not so. Writing a fair implementation is harder than an unfair implementation (and the extra logic required makes it less efficient), but certain algorithms might benefit from fairness, so we give both here.

To start with, we need to implement a Java queue class. This is an abstract data type that encapsulates the logic required to define an unbounded, unordered buffer of objects. Note that it is best to use generics here, otherwise the queue will only be able to pop objects of `Object` type, and this would mean we would need to upcast them, e.g. `((C) q.pop())`.

```
class Queue<T> {

        private List list = new LinkedList();

        public synchronized T push(T x) {
                list.append(x);
        }

        public synchronized T pop(T x) {
                return (T)list.remove(0);
        }
```

```
        public synchronized boolean nonEmpty() {
                return list.size()!=0;
        }

}

class FairQueue<T> {

        private List list = new LinkedList();

        public synchronized T push(T x) {
                list.append(x);
        }

        public synchronized T pop(T x) {
                return (T)list.remove((int)(Math.random()*list.size()));
        }

        public synchronized boolean nonEmpty() {
                return list.size()!=0;
        }

}
```

Using this class, we can create a queue object as a field for each method that needs one in a class. We can name these fields after the method that they are associated with in the $\mathcal{L}^{J*}$ program, because a field can have the same name as a method in a Java class. Thus we can translate `push` x `this` m into `this.`$m.push(x)$ and `pop this` m into `this.`$m.pop()$.

The translation of the `ndcond this` $(m_1^1 \ldots m_{n^1}^1, \mathbf{e}^1) \ldots (m_1^k \ldots m_{n^k}^k, \mathbf{e}^k)(\mathbf{e})$ statement is much harder. First we consider the unfair version:

`if (this.`$m_1^1.nonEmpty()$ `&& ... && this.`$m_{n^1}^1.nonEmpty()$`) {`
    e^1;
`} else if (this.`$m_1^2.nonEmpty()$ `&& ... && this.`$m_{n^2}^2.nonEmpty()$`) {`
    e^2;
`} else if ...`
    $\vdots$
`} else if (this.`$m_1^k.nonEmpty()$ `&& ... && this.`$m_{n^k}^k.nonEmpty()$`) {`
    e^k;
`} else {`
    e;
`}`

The above code will choose the first suitable pair that it sees, and execute e, the default expression, if there are none available.

The fair version is much more complicated because we cannot choose any particular order when we test the queues. One method is to use the pseudo-random number generator (which has uniform distribution) to get an offset between 1 and k. We can then scan the pairs starting at this offset and "wrapping" when we get to the end, i.e. we can scan at index $((i + offset)$ `%` $k) + 1$ where $0 \leq i < k$. In the following code we use a switch statement to achieve this:

```
int offset = (int)(Math.random() * k);
for (int i = 0 ; i < k ; i++) {
      switch (((i + offset) % k) + 1) {
            case 1 :
                  if (this.m_1^1.nonEmpty() && ... && this.m_{n^1}^1.nonEmpty()) {
                        e^1;
                        goto done :
                   } break;
            ⋮
            case k :
                  if (this.m_1^k.nonEmpty() && ... && this.m_{n^k}^k.nonEmpty()) {
                        e^k;
                        goto done :
                   } break;
      }
}
e;
done :
```

The general approach is rather complicated. It would be better for the programmer to tailor the algorithm for the specific case at hand. The expression being encoded may have an empty set of methods for some of the pairs, or only one pair, and thus be much easier to encode. We could represent this idea formally by defining behaviour-preserving optimisation rules that remove redundant code from applications of the algorithm above, but efficient implementations are not the subject of this report.

8.1.4 Guided tour of $\mathcal{L}^{J*}$

The formalism $\mathcal{L}^{J*}$ is very similar to $\mathcal{L}^{J}$, which was described in chapter 7, so we only consider the differences here.

Method signatures in $\mathcal{L}^{J*}$ specify the type for each of the sequence of argument variables, so for example t_1 is the type of x_1. In the source expressions, method calls now have a sequence of argument parameters.

Instead of having a single body for each method, like $\mathcal{L}^{J}$, in $\mathcal{L}^{J*}$ we have a set of methods.

The program tuple does not have fields, but instead has a mapping from class identifiers to sets of method identifiers. This represents the subset of class methods that have associated queues. We can tell the type of the values that will be stored in these queues, because it is always the same as the type of the single argument of the method. Thus a class has a set of associated method members, and a subset of this will also have queues attached. Operations on these queues refer to them by the identifier of the method to which they are attached.

The source expressions contain the constructs `push` x `this` m, `pop this` m, and `ndcond this` ..., that are operations on the queues contained within an object.

The heap represents queues in the same way that the chorded formalisations do. The run-time expressions include the queue operations, but here they have concrete values rather than variable identifiers. When source expressions are converted into run-time

expressions during method invocation, the substitution changes the `this` into some ι, and the argument x into some value.

Of the semantics rules that $\mathcal{L}^{J*}$ shares with $\mathcal{L}^J$, most remain unchanged. Object construction (New) is different because there are no fields, and instead we are initialising a mapping to empty queues, however this is the same as in $\mathcal{L}^S$ and $\mathcal{L}^A$. Method invocation (Inv) is different because not only does it choose an arbitrary body to invoke (i.e. there is non-determinism), but it also has to substitute the sequence of argument parameters, rather than just a single argument.

The main change is the new rules (Push), (Pop), and (NdCond). The mechanisms within (Push) and (Pop) are borrowed from the chord semantics of $\mathcal{L}^S$ and $\mathcal{L}^A$, they just literally look up the queues representation from the appropriate instance, and either append or remove an element from the appropriate queue. The rule (Pop) returns the value it pops, whereas (Push) returns `voidval`.

The translation to $\mathcal{L}^{J*}$ never attempts to pop an element from an empty queue, we can be sure of this because of the mutual exclusion and the test for emptiness. The behaviour of the semantics in this case is therefore not important. With the rules as they stand, a thread that attempts to pop an empty queue will block indefinitely, without giving up any of its locks, so will most likely cause a deadlock. Perhaps we should define some kind of exception to be raised in this event, to make the error condition more explicit, but this has not been done in this report.

The rule (NdCond) is more complex. The expression will reduce to one of $\{e_1 \dots e_k, e\}$. Recall that we call a pair $(m^i_1 \dots m^i_{n^i}, e^i)$ "suitable" if the queues associated with $m^i_1 \dots m^i_{n^i}$ are non-empty. Thus the top case within the (NdCond) rule will apply for some i, if the i^{th} pair is suitable. Thus, the bottom case only applies if there is no i such that the i^{th} pair is suitable, and thus the expression chosen is the "default" one. Otherwise the expression returned is e_i, and the choice out of the set of possible suitable values for i is arbitrary.

The judgement (WFClass) ensures that all the methods that have associated queues actually exist in the class, and return `void`. They do not have to return `void` for any reason of type safety, but since in the translation from $\mathcal{L}^S$, every queue belongs to an asynchronous method, and all asynchronous methods have a return type of `void`, we can afford to make this restriction.

The type rules for the queue operations ensure that the queues being referenced exist within the appropriate class. The rules (STPush) and (STPop) also use $\mathcal{M}(P^{J*}, c, m)$ to get the type of the elements that will be stored in the queue. The rule (STNdCond) ensures that all the expressions have the same type. Because the `ndcond` expression itself resolves to one of these expressions, it too has the same type as all the sub-expressions.

8.2 Definition of the translation φ_{SJ*}

8.2.1 Example

To demonstrate how the translation works, we first consider an example $\mathcal{L}^S$ program. The program has one class, and one fairly general synchronous chord. We translate the program into a valid $\mathcal{L}^{J*}$ program without changing its behaviour or structure. Here is the program:

Programs:

$$
\begin{array}{rcll}
P^{J*} \in \mathcal{L}^{J*} & = & Id_c \times Id_m \to Methsig & \text{(Type signatures)} \\
& \times & Id_c \times Id_m \to \mathbb{P}(SrcExpr) & \text{(Method bodies)} \\
& \times & Id_c \to \mathbb{P}(Id_m) & \text{(Queues)} \\
& \times & Id_c \to Id_c & \text{(Superclass)} \\
Methsig & ::= & t_r\ m(t_1 \dots t_n) & \\
t \in Types & ::= & c \mid \texttt{void} & \\
\mathbf{e} \in SrcExpr & ::= & \texttt{null} \mid \texttt{voidval} \mid \texttt{this} \mid x \mid \texttt{new}\ c \mid \mathbf{e}.m(\mathbf{e}_1 \dots \mathbf{e}_n) & \\
& \mid & \mathbf{e}\ ;\ \mathbf{e} \mid \texttt{spawn}\ \mathbf{e} \mid \texttt{lock}\ \mathbf{e} \mid \texttt{unlock}\ \mathbf{e} \mid \texttt{wait}\ \mathbf{e} & \\
& \mid & \texttt{notify}\ \mathbf{e} \mid \texttt{notifyAll}\ \mathbf{e} \mid \texttt{push}\ x\ \texttt{this}\ m \mid \texttt{pop this}\ m & \\
& \mid & \texttt{ndcond this}\ ((m_1^1 \dots m_{n^1}^1), \mathbf{e}_1) \dots ((m_1^k \dots m_{n^k}^k), \mathbf{e}_k)(\mathbf{e}) & \\
m \in Id_m & & c \in Id_c \qquad x \in Id_a &
\end{array}
$$

Runtime objects:

$$
\begin{array}{rcl}
h \in Heap & = & \mathbb{N} \to (Id_c \times (Id_m \to Multiset(Val))) \\
e \in RunExpr & ::= & v \mid \texttt{new}\ c \mid e.m(e_1 \dots e_n) \mid e\ ;\ e \mid \texttt{spawn}\ e \\
& \mid & \texttt{lock}\ e \mid \texttt{unlock}\ e \mid \texttt{wait}\ e \mid \texttt{notify}\ e \\
& \mid & \texttt{notifyAll}\ e \mid \texttt{push}\ v\ \iota\ m \mid \texttt{pop}\ \iota\ m \\
& \mid & \texttt{ndcond}\ \iota\ ((m_1^1 \dots m_{n^1}^1), e_1) \dots ((m_1^k \dots m_{n^k}^k), e_k)(e) \\
& \mid & \texttt{locked}\ \iota\ e \mid \texttt{waiting}\ \iota \\
v \in Val & ::= & \texttt{null} \mid \texttt{voidval} \mid \iota \\
\iota \in \mathbb{N} & & \text{(Addresses)} \\
E[\cdot] & ::= & E[\cdot].m(e_1 \dots e_n) \mid \iota.m(v_1 \dots v_i, E[\cdot], e_1 \dots e_j) \\
& \mid & E[\cdot]\ ;\ e \mid v\ ;\ E[\cdot] \mid \texttt{lock}\ E[\cdot] \mid \texttt{unlock}\ E[\cdot] \\
& \mid & \texttt{wait}\ E[\cdot] \mid \texttt{notify}\ E[\cdot] \mid \texttt{notifyAll}\ E[\cdot] \\
& \mid & \texttt{locked}\ \iota\ E[\cdot]
\end{array}
$$

$$
\mathcal{M}(P^{J*}, c, m) = P^{J*}{\downarrow_1}(c, m) \qquad \mathcal{M}\mathit{eths}(P^{J*}, c, m) = P^{J*}{\downarrow_2}(c, m)
$$

$$
\mathcal{Q}(P^{J*}, c) = P^{J*}{\downarrow_3}(c) \qquad \mathcal{S}\mathit{up}(P^{J*}, c) = P^{J*}{\downarrow_4}(c)
$$

Figure 8.1: Syntax of $\mathcal{L}^{J*}$.

$$\frac{P^{J*} \vdash e, h \rightsquigarrow e', h'}{P^{J*} \vdash e_1 \dots e_n, e, h \rightsquigarrow e_1 \dots e_n, e', h'} \quad \text{(Run)} \qquad \frac{h(\iota) = \mathcal{U}df}{P^{J*} \vdash E[\texttt{new } c], h \rightsquigarrow E[\iota], h[\iota \mapsto [\![c \| \lambda m.\emptyset]\!]]} \quad \text{(New)}$$

$$\frac{h(\iota) = [\![c \| \_]\!], \qquad \mathbf{e} \in \mathcal{M}eths(P, c, m)}{P^{J*} \vdash E[\iota.m(v_1 \dots v_n)], h \rightsquigarrow E[\mathbf{e}[v_1/x_1 \dots v_n/x_n,\ \iota/\texttt{this}]], h} \quad \text{(Inv)}$$

$$\frac{\begin{array}{c}\forall i \in \{1 \dots n\}. \nexists E'. e_i = E'[\texttt{locked } \iota \ \_] \\ \nexists E'. E[\texttt{lock } \iota] = E'[\texttt{locked } \iota \ \_]\end{array}}{P^{J*} \vdash e_1 \dots e_n, E[\texttt{lock } \iota], h \rightsquigarrow e_1 \dots e_n, \texttt{locked } \iota \ E[\texttt{voidval}], h} \quad \text{(Lock)}$$

$$\frac{}{P^{J*} \vdash E_1[\texttt{locked } \iota \ E_2[\texttt{unlock } \iota]]], h \rightsquigarrow E_1[E_2[\texttt{voidval}]], h} \quad \text{(Unlock)}$$

$$\frac{}{P^{J*} \vdash E_1[\texttt{locked } \iota \ E_2[\texttt{wait } \iota]]], h \rightsquigarrow E_1[E_2[\texttt{waiting } \iota]], h} \quad \text{(Wait)}$$

$$\frac{}{\begin{array}{r}P^{J*} \vdash e_1 \dots e_n, E_1[\texttt{waiting } \iota], E_2[\texttt{locked } \iota \ E_3[\texttt{notify } \iota]], h \rightsquigarrow \\ e_1 \dots e_n, E_1[\texttt{lock } \iota], E_2[\texttt{locked } \iota \ E_3[\texttt{voidval}]], h\end{array}} \quad \text{(Notify)}$$

$$\frac{\forall i \in \{1 \dots n\}. \nexists E'. e_i = E'[\texttt{waiting } \iota]}{\begin{array}{r}P^{J*} \vdash e_1 \dots e_n, E_1[\texttt{locked } \iota \ E_2[\texttt{notify } \iota]], h \rightsquigarrow \\ e_1 \dots e_n, E_1[\texttt{locked } \iota \ E_2[\texttt{voidval}]], h\end{array}} \quad \text{(NotifyNone)}$$

$$\frac{\forall i \in \{1 \dots n\}. e'_i = \begin{cases} E'[\texttt{lock } \iota] & \text{if } e_i = E'[\texttt{waiting } \iota] \\ e_i & \text{otherwise} \end{cases}}{\begin{array}{r}P^{J*} \vdash e_1 \dots e_n, E_1[\texttt{locked } \iota \ E_2[\texttt{notifyAll } \iota]], h \rightsquigarrow \\ e'_1 \dots e'_n, E_1[\texttt{locked } \iota \ E_2[\texttt{voidval}]], h\end{array}} \quad \text{(NotifyAll)}$$

$$\frac{h(\iota) = [\![c \| qs]\!]}{P^{J*} \vdash E[\texttt{push } v \ \iota \ m], h \rightsquigarrow E[\texttt{voidval}], h[\iota \mapsto [\![c \| qs[m \mapsto qs(m) \cup \{v\}]]\!]]} \quad \text{(Push)}$$

$$\frac{h(\iota) = [\![c \| qs]\!], \qquad v \in qs(m)}{P^{J*} \vdash E[\texttt{pop } \iota \ m], h \rightsquigarrow E[v], h[\iota \mapsto [\![c \| qs[m \mapsto qs(m) \backslash \{v\}]]\!]]} \quad \text{(Pop)}$$

$$\frac{\begin{array}{c}h(\iota) = [\![c \| qs]\!] \\ e' = \begin{cases} e_i & \text{if } i \in \{1 \dots k\}. \forall j \in \{1 \dots n_k\}. qs(m^i_j) \neq \emptyset \\ e & \text{otherwise} \end{cases}\end{array}}{P^{J*} \vdash E[\texttt{ndcond } \iota \ ((m^1_1 \dots m^1_{n^1}), e_1) \dots ((m^k_1 \dots m^k_{n^k}), e_k)(e)], h \rightsquigarrow E[e'], h} \quad \text{(NdCond)}$$

$$\frac{}{P^{J*} \vdash e_1 \dots e_n, E[\texttt{spawn } e], h \rightsquigarrow e_1 \dots e_n, E[\texttt{voidval}], e, h} \quad \text{(Spawn)}$$

Figure 8.2: Semantics of $\mathcal{L}^{J*}$.

Well-formedness:

$$\frac{\forall c.\mathcal{S}up(P^{J*},c) \neq \mathcal{U}df \Longrightarrow P^{J*} \vdash c}{\vdash P^{J*}} \quad \text{(WFProg)}$$

$$\frac{\begin{array}{l} P^{J*} \vdash \mathcal{S}up(P^{J*},c) \diamond_{cl} \\ \forall m.\mathcal{M}(P,\mathcal{S}up(P^{J*},c),m) = t_r\ m(t_1 \dots t_n) \Longrightarrow \\ \qquad \exists t'_r \sqsubseteq t_r, t'_1 \sqsupseteq t_1 \dots t'_n \sqsupseteq t_n.\mathcal{M}(P^{J*},c,m) = t'_r\ m(t'_1 \dots t'_n) \\ \forall m, \mathbf{e} \in \mathcal{M}eths(P^{J*},c,m) \\ \qquad \mathcal{M}(P^{J*},c,m) = t_r\ m(t_1 \dots t_n), \\ \qquad P^{J*}, [x_1 \mapsto t_1 \dots x_n \mapsto t_n, \texttt{this} \mapsto c] \vdash \mathbf{e} : t_r \\ \forall m \in \mathcal{Q}(P^{J*},c).\mathcal{M}(P^{J*},c,m) = \texttt{void}\ m(\_) \end{array}}{P^{J*} \vdash c} \quad \text{(WFClass)}$$

$$\frac{\mathcal{S}up(P^{J*},c) \neq \mathcal{U}df \quad \vee \quad c = \texttt{Object}}{P^{J*} \vdash c \diamond_{cl}} \quad \text{(IsClass)}$$

Source type rules: (for (STLock), (STUnlock), (STNotify) and (STNotifyAll) see (STWait))

$$\frac{v \in \{\texttt{this}\} \cup Id_a}{P^{J*},\Gamma \vdash v : \Gamma(v)} \quad \text{(STVar)} \qquad \frac{P^{J*} \vdash c \diamond_{cl}}{P^{J*},\Gamma \vdash \texttt{null} : c} \quad \text{(STNull)}$$

$$\frac{}{P^{J*},\Gamma \vdash \texttt{voidval} : \texttt{void}} \quad \text{(STVoid)} \qquad \frac{P^{J*} \vdash c \diamond_{cl}}{P^{J*},\Gamma \vdash \texttt{new}\ c : c} \quad \text{(STNew)}$$

$$\frac{\begin{array}{c} P^{J*},\Gamma \vdash \mathbf{e} : c \\ \forall i \in \{1 \dots n\}.P^{J*},\Gamma \vdash \mathbf{e}_i : t_i \\ \mathcal{M}(P^{J*},c,m) = t_r\ m(t_1 \dots t_n) \end{array}}{P^{J*},\Gamma \vdash \mathbf{e}.m(\mathbf{e}_1 \dots \mathbf{e}_n) : t_r} \quad \text{(STInv)} \qquad \frac{\begin{array}{c} P,\Gamma \vdash \mathbf{e}_1 : t_1 \\ P,\Gamma \vdash \mathbf{e}_2 : t_2 \end{array}}{P,\Gamma \vdash \mathbf{e}_1\ ;\ \mathbf{e}_2 : t_2} \quad \text{(STSeq)}$$

$$\frac{\begin{array}{c} P^{J*},\Gamma \vdash \mathbf{e} : c \\ P^{J*},\Gamma \vdash \mathcal{S}up(P^{J*},c) = c' \end{array}}{P^{J*},\Gamma \vdash \mathbf{e} : c'} \quad \text{(STSub)} \qquad \frac{P^{J*},\Gamma \vdash \mathbf{e} : \texttt{void}}{P^{J*},\Gamma \vdash \texttt{spawn}\ \mathbf{e} : \texttt{void}} \quad \text{(STSpawn)}$$

$$\frac{P^{J*},\Gamma \vdash \mathbf{e} : c}{P^{J*},\Gamma \vdash \texttt{wait}\ \mathbf{e} : \texttt{void}} \quad \text{(STWait)} \qquad \frac{\begin{array}{c} \Gamma(x) = t, \qquad \Gamma(\texttt{this}) = c \\ m \in \mathcal{Q}(P^{J*},c) \\ \mathcal{M}(P^{J*},c,m) = \texttt{void}\ m(t) \end{array}}{P^{J*},\Gamma \vdash \texttt{push}\ x\ \texttt{this}\ m : \texttt{void}} \quad \text{(STPush)}$$

$$\frac{\begin{array}{l} \Gamma(\texttt{this}) = c \\ \forall i \in \{1 \dots k\}. \\ \qquad P^{J*},\Gamma \vdash e^i : t \\ \qquad \forall j \in \{1 \dots n_i\}.m^i_j \in \mathcal{Q}(P^{J*},c) \\ P^{J*},\Gamma \vdash \mathbf{e} : t \end{array}}{\begin{array}{r} P^{J*},\Gamma \vdash \texttt{ndcond this}\ (m^1_1 \dots m^1_{n^1},\mathbf{e}^1) \\ \dots (m^k_1 \dots m^k_{n^k},\mathbf{e}^k)(\mathbf{e}) : t \end{array}} \quad \text{(STNd)}$$

$$\frac{\begin{array}{c} \Gamma(\texttt{this}) = c \\ m \in \mathcal{Q}(P^{J*},c) \\ \mathcal{M}(P^{J*},c,m) = \texttt{void}\ m(t) \end{array}}{P^{J*},\Gamma \vdash \texttt{pop this}\ m : t} \quad \text{(STPop)}$$

Figure 8.3: Rules for well-formed $\mathcal{L}^{J*}$ programs.

```
class C {
    t_r m1(t1 m1_x) & async m2(t2 m2_x) & ... & async mn(tn mn_x) {
        e
    }
}
```

This is converted into the following code: Note the new method `m_b` is used to hold the body of the original chord, while all the other methods play the role of the chord semantics rules (InvS) and (InvA).

The method `m1` checks whether the required items are on the queues (using the semantics of `ndcond this`). If there are queued values available then it invokes the body and returns the result. This is achieved by calling the method `m_b`, which contains the chord body, and the values on the queue are held in this methods arguments. In a real language it would not be necessary to create this additional method since there would be local variables available. In our abstraction, however, there is no other way.

If there is not a full complement of queued values it executes `wait this`, until another thread calls one of the asynchronous methods, whereupon it is woken up and recursively executes `m1` to check the queues again.

The asynchronous methods `m2` to `mn` place values onto the queues and notify any threads that need to synchronise on these queues, that they need to wake up and check again. All implementations of the chord semantics rules (InvS) and (InvA) are protected by the mutex so that threads do not encounter an inconsistent state of the queues.

```
class C {
    t_r m1(t1 x_1) {
        lock this;
        ndcond (m2 ... mn, m_b(pop this m1, ..., pop this mn, x_1)
               (wait this; unlock this; m1(x_1))
    }

    void m2(t2 m2_x) {
        lock this;
        push x this m2;
        notifyAll this;
        unlock this
    }
     ...
    void mn(tn mn_x) {
        lock this;
        push x this mn;
        notifyAll this;
        unlock this
    }

    void m_b(t1 x_1, t2 x_2, ..., tn x_n) {
        unlock this;
        e[x_1/m1_x ... x_n/mn_x]
    }
}
```

8.2.2 Translation

The encoding is given in figure 8.4. Because the encoding is non-deterministic (in the choice of the names of new methods, and also because we need to define some ordering on the methods in a chord) we collect this flexibility into a single "decision" function

$$\delta : \Delta = (Id_c \times Id_m \times Chord^S) \to (Id_m \times (\mathbb{N} \to Id_m))$$

and specify this as an argument to φ_{SJ*}.

For each synchronous method m in class c, and associated chord ch, $\delta(c, m, ch) = (m_b, X)$ tells us what method will contain the body of the synchronous chord, and also gives us a mapping from an index, to the methods in the chord, so that we can identify specific methods by their number.

This is required because otherwise when we take $\{m_1 \dots m_n\} = ch\downarrow_2$, the ordering of $m_1 \dots m_n$ is arbitrary. Instead we use $X(1) \dots X(n)$ which is defined at the top level of the translation, so we know it will be consistent. In $\mathcal{L}^S$, arguments were identified with $m_1\_\mathtt{x} \dots m_n\_\mathtt{x}$, i.e. named after their respective methods. In $\mathcal{L}^{J*}$ we have only a single method per method body, we must identify arguments with an index $x_1 \dots x_n$. Without X, the arguments lose their identity when encoded into a single method. We must be sure that we correctly substitute the arguments in the chord body.

The result of $\varphi_{SJ*}(P^S, \delta)$ is only well-defined when δ is well-formed, i.e. when $P^S \vdash \delta$ as defined below:

$$\begin{aligned} P^S \vdash \delta \iff\ & \forall c, m, ch \in \mathcal{SChs}(P^S, c, m)\ . \\ & \quad \text{let } \delta(c, m, ch) = (m_b, X) \\ & \quad \mathcal{M}(P^S, c, m_b) = \mathcal{Udf}\ ,\ \forall m'_b = \delta(c', m', ch')\downarrow_1 . c' = c, m' = m, ch' = ch \\ & \quad X \text{ is a bijection from } \{1 \dots |ch\downarrow_1|\} \text{ to } ch\downarrow_1 \\ & \forall c, m, ch . \delta(c, m, ch) \neq \mathcal{Udf} \implies \\ & \quad ch \in \mathcal{SChs}(P^S, c, m) \end{aligned}$$

8.2.3 Properties of the translation

These properties are not interesting results in the broader scope of this report, but are presented here because they reflect the characteristics of this translation, and thus help us understand it. If $P^{J*} = \varphi_{SJ*}(P^S, \delta)$:

- $|\mathcal{Methods}(P^{J*}, c, m)| \leq 2$ This is used purely to capture the idea that when a method is a synchronous part and an asynchronous part of chords in a class, it is non-deterministic which behaviour is chosen.

- $\mathcal{M}(P^S, c, m) \neq \mathcal{Udf} \implies \mathcal{M}(P^S, c, m) = t_r\ m(t_a)$ so the number of arguments in $\mathcal{M}(P^{J*}, c, m)$ is 1.

$\varphi_{SJ*}(P^S, \delta) = P^{J*}$ if and only if

$$\mathcal{M}(P^{J*}, c, m) = \begin{cases} \mathcal{M}(P^S, c, m) & \text{if } \mathcal{M}(P^S, c, m) \neq \mathcal{U}df \\ t_r\ m(t_1 \dots t_n, t_s) & \text{if } \exists ch, m_s, c \sqsubseteq c'\ .\ \delta(c', m_s, ch) = (m, X),\ n = |ch{\downarrow_1}|, \\ & \quad \mathcal{M}(P^S, c', m_s) = t_r\ m_s(t_s), \\ & \quad \forall i \in \{1 \dots n\}.\mathcal{M}(P^S, c', X(i)) = \texttt{void}\ X(i)(t_i) \end{cases}$$

$$\begin{aligned} \mathcal{M}eths(P^{J*}, c, m) \quad = \quad & \{\ \texttt{unlock this}; ch{\downarrow_2}[x_1/X(1)\_\texttt{x} \dots x_n/X(n)\_\texttt{x}, x_{n+1}/m_s\_\texttt{x}] \\ & \qquad\qquad \mid \exists m_s, ch, X.\delta(c, m_s, ch) = (m, X), n = |ch{\downarrow_1}|\ \} \\ \cup \quad & \{\ \texttt{lock this}; \texttt{push}\ x\ \texttt{this}\ m; \texttt{unlock this} \mid \text{if } m \in \mathcal{Q}(P^S, c)\ \} \\ \cup \quad & \{\ \texttt{lock this}; \\ & \quad \texttt{ndcond this}\ (X^1(1) \dots X^1(n^1), m_b^1(\texttt{pop this}\ X^1(1), \dots, \texttt{pop this}\ X^1(n^1), x_1)) \\ & \qquad\qquad \vdots \\ & \qquad\qquad (X^k(1) \dots X^k(n^k), m_b^k(\texttt{pop this}\ X^k(1), \dots, \texttt{pop this}\ X^k(n^k), x_1)) \\ & \qquad\qquad (\texttt{wait this}; \texttt{unlock this}; \texttt{this}.m(x_1))\ \mid \\ & \quad \text{if } \mathcal{SC}hs(P^S, c, m) = \{ch^1 \dots ch^k\},\ k \geq 1, \\ & \qquad \forall j \in \{1 \dots k\}.n^j = |ch^j{\downarrow_1}|,\ \delta(c, m, ch^j) = (m_b^j, X^j)\ \} \end{aligned}$$

$\mathcal{Q}(P^{J*}, c) = \mathcal{Q}(P^S, c)$

$\mathcal{S}up(P^{J*}, c) = \mathcal{S}up(P^S, c)$

Figure 8.4: Translation $\varphi_{SJ*} : (\mathcal{L}^S \times \Delta) \to \mathcal{L}^{J*}$

8.3 Preservation of structure (proof)

We first formally define the property of structure preservation for our programs P^S and P^{J*} where $P^{J*} = \varphi_{SJ*}(P^S, \delta)$. We refer to section 3.1.2, where the contribution of [5] is put into the context of our object-oriented formalisations. In order for the translation to be structure-preserving, it must satisfy:

- All syntactic constructs of the source expressions of $\mathcal{L}^S$ are also present in $\mathcal{L}^{J*}$ are the five constructs that implement mutual exclusion. Thus, source expressions must not be modified in the translation.

  The synchronous chord bodies of P^S are placed into methods of P^{J*}, unchanged except for the renaming of arguments which is explicitly allowed in section 3.1.4.

- Method signatures are common to $\mathcal{L}^S$ and $\mathcal{L}^{J*}$ ($\mathcal{L}^{J*}$ allows multiple arguments, but this is an extension of $\mathcal{L}^S$'s single arguments), so they must not be altered, although we do allow the addition of new methods.
  $\forall c, m. \mathcal{M}(P^S, c, m) = msig \Rightarrow \mathcal{M}(P^{J*}, c, m) = msig$

  From $P^S \vdash \delta$ we know that the two cases for the definition of $\mathcal{M}(P^{J*}, c, m)$ in φ_{SJ*} are exclusive, so when $\mathcal{M}(P^S, c, m) = msig$ i.e. $\mathcal{M}(P^S, c, m) \neq \mathcal{U}df$, we know that $\mathcal{M}(P^{J*}, c, m) = \mathcal{M}(P^S, c, m) = msig$. □

- Synchronous chords are present in $\mathcal{L}^S$ but not in $\mathcal{L}^{J*}$, so they must be replaced with a macro, and each body must be used exactly once in the macro.

  The first case of $\mathcal{M}eths(P^{J*}, c, m)$ in φ_{SJ*} defines a method body containing $ch{\downarrow_2}$ where δ defines m to be the method that holds the body of the synchronous chord ch of method m_s in class c. Since every such synchronous chord has its own unique method for holding its body, we know that the body will only be used once, and we can immediately see it is not changed except for the renaming of arguments which is allowed as explained in section 3.1.4. □

- The feature of classes is common to $\mathcal{L}^S$ and $\mathcal{L}^{J*}$, so the structure of classes must be preserved, although we allow the addition of new classes:
  $\forall c. \mathcal{S}up(P^S, c) = c' \Longrightarrow \mathcal{S}up(P^{J*}, c) = c'$

  We can prove this by inspection of φ_{SJ*}, which actually has the stronger property that no classes are added or removed. □

8.4 Preservation of well-formedness (proof)

We need to establish that the well-formedness of a program is preserved when the program is translated, this is the notion of preservation of programness in [5]. Let $P^{J*} = \varphi_{SJ*}(P^S, \delta)$. We must show that $\vdash P^S \Longrightarrow \vdash P^{J*}$.

Lemma 8.4.1 *If* $P^{J*} = \varphi_{SJ*}(P^S, \delta)$ *then* $P^S \vdash c \diamond_{cl} \Longrightarrow P^{J*} \vdash c \diamond_{cl}$

Proof:

$$
\begin{array}{lll}
P^S \vdash c \diamond_{cl} & \Longrightarrow & \mathcal{S}up(P^S, c) \neq \mathcal{U}df \vee c = \texttt{Object} \quad \text{(IsClass)} \\
& \Longrightarrow & \mathcal{S}up(P^{J*}, c) \neq \mathcal{U}df \vee c = \texttt{Object} \quad \text{(Def } \varphi_{SJ*}\text{)} \\
& \Longrightarrow & P^{J*} \vdash c \diamond_{cl} \quad \text{(IsClass)}
\end{array}
$$

Lemma 8.4.2 *Method type signatures are preserved over inheritance in the translated program: If* $\vdash P^S$ *and* $P^{J*} = \varphi_{SJ*}(P^S, \delta)$ *then*
$\forall m.\mathcal{M}(P^{J*}, \mathcal{S}up(P^{J*}, c), m) = t_r\ m(t_1 \ldots t_n) \Longrightarrow$
$\exists t'_r \sqsubseteq t_r, t'_1 \sqsupseteq t_1 \ldots t'_n \sqsupseteq t_n\ .\ \mathcal{M}(P^{J*}, c, m) = t'_r\ m(t'_1 \ldots t'_n)$.
Proof:

Let $\vdash P^S$ and $\mathcal{M}(P^{J*}, \mathcal{S}up(P^{J*}, c), m) = t_r\ m(t_1 \ldots t_n)$. We know from the definition of φ_{SJ*} that there are two possibilities:

- $\mathcal{M}(P^S, \mathcal{S}up(P^S, c), m) = t_r\ m(t_a) \neq \mathcal{U}df$, $n = 1$, in which case from $P^S \vdash c$ we know that $\exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a.\mathcal{M}(P^S, c, m) = t'_r\ m(t'_a)$, and from the definition of φ_{JS*} we know that $\mathcal{M}(P^{J+}, c, m) = t'_r\ m(t'_a)$ □

- $\exists ch, m_s, \mathcal{S}up(P^{J*}, c) \sqsubseteq c'.\delta(c', m_s, ch) = (m, X)$, etc. Clearly the same case will apply for $\mathcal{M}(P^{J*}, c, m)$ because $c \sqsubseteq \mathcal{S}up(P^{J*}, c) \sqsubseteq c'$, and thus the subclass's method will have the same type. □

Lemma 8.4.3 *Well-typedness of expressions is preserved over the translation: If* $P^{J*} = \varphi_{SJ*}(P^S, \delta)$, *then* $P^S, \Gamma \vdash \boldsymbol{e}^S : t \Longrightarrow P^{J*}, \Gamma \vdash \boldsymbol{e}^S : t$
Proof: Induction over the structure of derivations.

- STVar, STVoid: Trivial. □

- STNull, STNew: use lemma 7.5.1. □

- STInv: From the definition of φ_{JS*},
$\mathcal{M}(P^S, c, m) = t\ m(t_a) \neq \mathcal{U}df \Longrightarrow \mathcal{M}(P^{J*}, c, m) = t\ m(t_a)$. Using this and the induction hypothesis we can show $P^{J*}, \Gamma \vdash \mathbf{e}_1.m(\mathbf{e}_2) : t$. □

- STSeq, STSpawn: Use the induction hypothesis. □

- STSub: Use $\mathcal{S}up(P^{S+}, c) = \mathcal{S}up(P^J, c)$ from definition of φ_{JS+}, and the induction hypothesis. □

Lemma 8.4.4 *If* $P, \Gamma \vdash \boldsymbol{e} : t$ *and* $\Gamma'(v[x_1/m_1\_\boldsymbol{x} \ldots x_n/m_n\_\boldsymbol{x}]) = \Gamma(v)$ *for* $v \in \{x_1 \ldots x_n, \texttt{this}\}$, *then* $P, \Gamma' \vdash \boldsymbol{e}[x_1/m_1\_\boldsymbol{x} \ldots x_n/m_n\_\boldsymbol{x}] : t$
Proof: Induction over the structure of derivations.

- Only interesting case is (STVar), since all the other rules type other kinds of syntax, and ignore Γ. Because the changes to the expression being typed and the environment are compatible, we simply use the definition of Γ' to type the substitution of $\mathbf{e}$ to t. □

Lemma 8.4.5 $P^S \vdash c \Longrightarrow P^{J*} \vdash c$

Proof:

$$\begin{array}{lll} P^S \vdash c & \Longrightarrow & P^S \vdash \mathcal{S}up(P^S, c)\ \diamond_{cl} \quad \text{(WFClass)} \\ & \Longrightarrow & P^S \vdash \mathcal{S}up(P^{J*}, c)\ \diamond_{cl} \quad \text{(Def } \varphi_{SJ*}\text{)} \\ & \Longrightarrow & P^{J*} \vdash \mathcal{S}up(P^{J*}, c)\ \diamond_{cl} \quad \text{(Lemma 8.4.1)} \end{array}$$

$$\begin{array}{lll} P^S \vdash c & \Longrightarrow & \forall m.\mathcal{M}(P^{J*}, \mathcal{S}up(P^{J*}, c), m) = t_r\ m(t_a) \\ & & \Rightarrow \exists t'_r \sqsubseteq t_r, t'_a \sqsupseteq t_a\ .\ \mathcal{M}(P^{J*}, c, m) = t'_r\ m(t'_a) \quad \text{(Lemma 8.4.2)} \end{array}$$

For all bodies $\mathbf{e} \in \mathcal{M}eth(P^{J*}, c, m)$ either:

- $\exists m_s, ch, X.\delta(c, m_s, ch) = (m, X), n = |ch{\downarrow_1}|$ and $\mathbf{e} = \texttt{unlock this}; \mathbf{e}'$ where $\mathbf{e}' = ch{\downarrow_2}[x_1/X(1)\texttt{\_x} \ldots x_n/X(n)\texttt{\_x}, x_{n+1}/m_s\texttt{\_x}]$.
  - From $P^S \vdash \delta$ we know that $ch \in \mathcal{SCH}s(P^S, c, m_s)$ and for each $i \in \{1 \ldots n\}$, $X(i)$ is a distinct member of $ch{\downarrow_1}$. Thus from $P^S \vdash c$ we know that $P^S, \Gamma \vdash ch{\downarrow_2} : t_r$ where $\mathcal{M}(P^S, c, m_s) = t_r\ m_s(t_a)$, $\Gamma = [X(1)\texttt{\_x} \mapsto t_1 \ldots X(n)\texttt{\_x} \mapsto t_n, m_s\texttt{\_x} \mapsto t_s\texttt{this} \mapsto c]$ and $\forall i \in \{1 \ldots n\}.\mathcal{M}(P^S, c, X(i)) = \texttt{void}\ X(i)(t_i)$
  - From the definition of φ_{SJ*} we know that $\mathcal{M}(P^{S*}, c, m) = t_r\ m(t_1 \ldots t_n, t_s)$.
  - We can derive $P^{J*}, \Gamma' \vdash \mathbf{e}' : t_r$ where $\Gamma' = [x_1 \mapsto t_1 \ldots x_n \mapsto t_n, x_{n+1} \mapsto t_s, \texttt{this} \mapsto c]$ from the above typing $P^S, \Gamma \vdash ch{\downarrow_2} : t_r$, using lemma 8.4.2 and lemma 8.4.4.
  - Finally, `unlock this` trivially types to `void`, so the entire body types to $P^{J*}, \Gamma \vdash \mathbf{e} : t_r$. □
- $m \in \mathcal{Q}(P^S, c)$ in which case $\mathbf{e} = \texttt{lock this}; \texttt{push}\ x\ \texttt{this}\ m; \texttt{unlock this}$ and:
  - Because $m \in \mathcal{Q}(P^S, c)$, we know that $m \in \{m_1 \ldots m_n\}$ where $(\{m_1 \ldots m_n\}, \_) \in \mathcal{SCH}s(P^S, c, m_s)$ for some m_s. From $P^S \vdash c$, we know that $\mathcal{M}(P^S, c, m) = \texttt{void}\ m(t)$ for some t. By the definition of φ_{SJ*}, we know that $\mathcal{M}(P^{J*}, c, m) = \texttt{void}\ m(t)$.
  - Let $\Gamma = [x_1 \mapsto t, \texttt{this} \mapsto c]$. Trivially, $P^{J*}, \Gamma \vdash \texttt{lock this} : \texttt{void}$ and $P^{J*}, \Gamma \vdash \texttt{unlock this} : \texttt{void}$. Also, using (STPush), $P^{J*}, \Gamma \vdash \texttt{push}\ x\ \texttt{this}\ m : t$ since we know from the definition of φ_{SJ*} and $m \in \mathcal{Q}(P^S, c)$ that $m \in \mathcal{Q}(P^{J*}, c)$.

– Finally, we can type the composition of the three statements to `void`, i.e. $P^{J*}, \Gamma \vdash \mathbf{e} : \texttt{void}$. □

- $\mathbf{e} = \texttt{lock this}; \texttt{ndcond this} \ldots$ (see figure 8.4) where $\mathcal{SCHS}(P^{J*}, c, m) = \{ch_1 \ldots ch_k\}$, $\forall j \in \{1 \ldots k\}.(m_b^j, X^j) = \delta(c, m, ch_j), n^j = |ch^j{\downarrow_1}|$

  – Firstly we know from the fact that $\mathcal{SCHS}(P^S, c, m)$ is defined, that $\mathcal{M}(P^S, c, m) = t_r\ m(t_a)$, and thus from the definition of φ_{SJ*} that $\mathcal{M}(P^{J*}, c, m) = t_r\ m(t_a)$. We must show that $P^{J*}, \Gamma \vdash \mathbf{e} : t_r$ where $\Gamma = [x_1 \mapsto t_a, \texttt{this} \mapsto c]$. This is typed with STSeq, we need to type the `lock this` part, which is trivially `void`, and the `ndcond` part, which is more difficult:

  – We know from $P^S \vdash \delta$ that all the $X^j(i) \in ch^j$ so $X^j(i) \in \mathcal{Q}(P^S, c)$ and thus are also in $\mathcal{Q}(P^{J*}, c)$. From $P^S \vdash c$, we know that each $\mathcal{M}(P^S, c, X^j(i)) = \texttt{void}\ X(i)(t_i^j)$ and therefore $\mathcal{M}(P^{J*}, c, X^j(i)) = \texttt{void}\ X(i)(t_i^j)$ for some t_i^j. This means $P^{J*}, \Gamma \vdash \texttt{pop this}\ X^j(i) : t_i^j$ where $\Gamma = [x_1 \mapsto t_a, \texttt{this} \mapsto c]$.

  – From the definition of φ_{SJ*}, we can deduce that $\mathcal{M}(P^{J*}, c, m_b^j) = t_r\ m_b^j(t_1^j \ldots t_n^j, t_a)$. This means that for all $j \in \{1 \ldots k\}$ we can type: $P^{J*}, \Gamma \vdash m_b^j(\texttt{pop this}\ X^j(1) \ldots \texttt{pop this}\ X^j(n^j), x_1) : t_r$

  – We can also use the above information to show $P^{J*}, \Gamma \vdash \texttt{wait this}; \texttt{unlock this}; \texttt{this}.m(x_1) : t_r$.

  – This means we can use the (NdCond) rule to get $P^{J*}, \Gamma \vdash \mathbf{e}' : t_r$, and thus we can use (StSeq) to get $P^{J*}, \Gamma \vdash \mathbf{e} : t_r$. □

For all methods $m \in \mathcal{Q}(P^{J*}, c)$ we know from φ_{SJ*} that $m \in \mathcal{Q}(P^S, c)$, and this means $m \in \{m_1 \ldots m_n\}$ where $(\{m_1 \ldots m_n\}, \_) \in \mathcal{SCHS}(P^S, c, m)$ for some m. From $P^S \vdash c$, we know that $\forall i \in \{1 \ldots n\}.\mathcal{M}(P^S, c, m) = \texttt{void}\ m(\_)$

The above is sufficient to prove $P^{J*} \vdash c$. □

Theorem 8.4.6 $\vdash P^S \Longrightarrow \vdash P^{J*}$.

Proof:

$$
\begin{array}{lll}
\vdash P^S & \Longrightarrow & \forall c.\mathcal{Sup}(P^S, c) \neq \mathcal{Udf} \Rightarrow P^S \vdash c \qquad \text{(WFProg)} \\
& \Longrightarrow & \forall c.\mathcal{Sup}(P^{J*}, c) \neq \mathcal{Udf} \Rightarrow P^S \vdash c \qquad \text{(Def } \varphi_{SJ*}\text{)} \\
& \Longrightarrow & \forall c.\mathcal{Sup}(P^{J*}, c) \neq \mathcal{Udf} \Rightarrow P^{J*} \vdash c \quad \text{(Lemma 8.4.5)} \\
& \Longrightarrow & \vdash P^{J*} \qquad \text{(WFProg)}
\end{array}
$$

Chapter 9

Conclusion

9.1 Summary of the technical results of this project

We have defined a notion of structure-preservation of program translations (based on [5]). If a translation is structure-preserving, then it does not cause *unacceptable* change to the structure of programs. We believe this definition is a reasonable model of the perception a programmer might have, of whether the restructuring required when porting a program to a different language is acceptable or not.

We have defined four translations between five languages, that preserve program structure, validity, and behaviour. We have identified formal properties that define program validity (well-formedness) and structure, and proved that these properties are preserved over all the translations.

We have not proved the preservation of program behaviour over the translations, in other words we cannot be certain that the translated programs "do the same thing" as the original programs. We went some way towards this goal in section 5.4, but this is far from complete, and we have not looked at the other translations. However we do believe that the behaviour is preserved. In some way writing a translation is like writing a program, and programmers all over the world are writing programs that they believe to behave in a certain way, without formal verification. We assume for now that the translations do preserve behaviour, despite the lack of formal proof.

- We know from φ_{AS} (chapter 5) and φ_{SA} (chapter 6) that $\mathcal{L}^S$ and $\mathcal{L}^A$ are equally expressive. We believe this result will scale up to realistic languages containing many other features, such as Java and $C^\sharp$ (assuming those languages had chords). This means that a programmer can write their program using either asynchronous chords for thread creation, or using the `spawn` statement. Neither approach allows novel ways of structuring a program, and certainly neither expresses program *behaviour* that the other does not.

  Programmers do not often need to spawn new threads, and when they do, this process does not directly risk erroneous behaviour such as race conditions and deadlock, like other aspects of concurrent programming do. Thread creation is not directly related to synchronisation, so this result is not particularly interesting by itself.

- We assume that $\mathcal{L}^S$ and $\mathcal{L}^{S+}$ are equally expressive when expressing synchronisation,

because of the features present in $\mathcal{L}^{S+}$, fields do not add to the expressiveness[4], and integers are orthogonal to synchronisation so should not affect our results.

- We know from φ_{JS+} (chapter 7) that all programs in the language $\mathcal{L}^{J}$ can be rewritten into programs of $\mathcal{L}^{S+}$ without making radical changes to their structure and without changing their behaviour. This means that rewriting a program with chords instead of monitors will not require restructuring the program.

- We know from φ_{SJ+} (chapter 8) that all $\mathcal{L}^{S}$ programs, that is programs using synchronous chords, can be re-written into programs of $\mathcal{L}^{J*}$, i.e. they can be re-written to use monitors instead, without needing radical changes to their structure. This means that any programs that use chords can be re-written to use monitors, and thus chords do not allow programs to be written in new ways.

9.2 What does this mean for the programmer?

We know that chords are equally as expressive as monitors, but what does this mean for the programmer? Let us take an analogy. If we add a linked list implementation to a language's standard library, even though the language is expressive enough to represent lists by itself, we are not adding to the expressiveness of the language. We might still like to do this though, because code that needs lists will be more concise if it does not have to define lists itself. Just because monitors or chords are equally expressive, does not mean one should be discarded. So if we have both constructs in a language, how does the programmer know which one to choose?

9.2.1 Conciseness – an application suited to chords

What actually happens when we change a chorded program to a program using monitors? Our formal translations preserve structure, but we have not spoken of other factors such as conciseness. In section 2.3 we gave an example of a program ideally suited to chords, and its implementation in chords was very simple as a result. If we show how to express this program with monitors, we will see a "worst case" of the cruft that needs to be added by the translation.

This is presented in figure 9.1. It is not a direct application of φ_{SJ*}; it contains some obvious optimisations to make the code more concise, but the general idea behind the translation is the same. Note that we take some liberties here:

Firstly, there is no `lock()` or `unlock()` method associated with objects in Java, and even if we defined them ourselves (as described in section 2.2), it it would erroneously allow us to take the lock even though another thread was in a `synchronized` block. This is frustrating, but it is a problem with Java's implementation of monitors, not a problem with monitors themselves. There is no cause to switch to chords just because one language has a problematic implementation of monitors.

What Java should have is a unified mutex system, where there is one mutex per object, which is accessed both with structured synchronisation (`synchronized`) and unstructured synchronisation (i.e. the fundamental `lock()` and `unlock()` methods are exposed). If

```
public abstract class ActiveObject extends Thread {
        abstract protected void processMessage();
        void run () {
                while (!done) { this.processMessage(); }
        }
}

class StockServer extends ActiveObject {

        void processMessage() {
                lock();
                if (!q_addClient.empty()) {
                        Client c = q_addClient.pop();
                        unlock();
                        /* process addClient message */
                } else if (!q_wireQuote.empty()) {
                        Quote q = q_wireQuote.pop();
                        unlock();
                        /* process wireQuote message */
                } else if (q_closeDown > 0) {
                        q_closeDown--;
                        unlock()
                        /* process closeDown message */
                } else {
                        wait();
                        unlock();
                        processMessage();
                }
        }


        private List q_addClient = new LinkedList();
        synchronized void addClient(Client c) {
                q_addClient.add(c);
                notify();
        }

        private List q_wireQuote = new LinkedList();
        synchronized void wireQuote(Quote q) {
                q_wireQuote.add(q);
                notify();
        }

        private int q_closeDown = 0;
        synchronized void closeDown() {
                q_closeDown++;
                notify();
        }
}
```

Figure 9.1: Monitor translation of the active object example

there is more than one underlying implementation of the mutex, each with different performance in different circumstances, then the programmer could specify their choice at the top of the class. Instead Java has a mutex in each object that is accessed with `synchronized`, and also mutexes in the standard library that can be constructed and controlled by method calls. The two approaches are completely disjoint.

The effect of this is that if a program can elegantly use one or other of the techniques (structured or unstructured), but cannot access the advantages of both. We can speculate that this situation has developed due to the incremental improvements of the Java language. The Java language designers most likely wanted to avoid overhauling the implementation of `synchronized` so they added the more efficient monitors to the Java library, which is accessed with methods, and is therefore unstructured. We are idealistic here, by assuming that the languages of the future will not have a backlog of legacy compatibility to aversely affect their design.

The second liberty we take, is that Java allows us to interrupt calls to `wait()`, and this causes an exception to be thrown. This means we have to deal with this exception in our code, or it will not be well-typed. We omit this in the above translation since our consideration of chords does not include such details as interrupting the blocking of threads that call synchronous methods.

Taking into consideration that the active object example is very well suited to Java, we can count the amount of extra lines in the monitor translation. There are nine lines used for synchronisation, and thirteen used for implementing the buffer that is implicit with chords. If we were to choose between the two programs, we would naturally say the conciseness of the first makes it a better program. In short, there is less to go wrong.

9.2.2 Conciseness – an application suited to monitors

What about an application that is not so suited to chords? The active object example does not care about the ordering of messages it receives, it also does not require the buffers to be bounded. This is very convenient because this is exactly the semantics that chords offer. We should consider a program, the behaviour of which is precisely what the semantics of monitors provides.

Suppose we have a system of messages where it is more important that the latency of the system is low, than all the messages get processed. If the processing resources are not sufficient to process a message, it is discarded. This would by typical for media applications which have tough realtime requirements, or any system regularly sensing its environment, such as a ship's tactical radar, or a single client in a multi-player game. Let us take the radar case for illustration:

A ship's radar continually scans all the sky around the ship, it produces a report on the status of large moving objects within a certain range, and dispatches this report to an array of processing hardware. We can imagine the report as a two-dimensional array of pixels, like a video picture on a television. The processing hardware interprets this image and decides whether there are any threats, for example it can work out the velocity of objects by matching them with objects from previous "frames" and thus calculating how far they have moved.

It is essential that the most recent frames are processed quickly. Because of storage requirements, only one frame can be stored at a time so this frame is overwritten if a

new report is available. The code in this example could use an arbitrary finite buffer, but this would be more complex. It is not uncommon for safety critical and military projects to place concrete bounds on the size of buffers, so that the system's dynamic memory use is more predictable. By increasing the number of computers that process the frames, we can avoid discarding so many frames, which increases the accuracy of the computers' conclusion.

We model this with threads. The radar itself (that generates the messages) is a thread that continually attempts to push messages into a buffer. The ship's computers are a set of threads that continually poll the buffer. Each computer receives a message and processes it independently from the other computers.

```
class Radar {
        run() {
                while (true) {
                        /* scan environment */
                        buffer.push(report);
                }
        }
}
class Computer {
        run() {
                while (true) {
                        report = buffer.pop();
                        /* process report, some comms with other systems */
                        /* control weapons systems */
                }
        }
}
```

The question is now how to implement the buffer. It must be synchronised so that the radar and the computers do not interfere, it must not store more than one report. Calls to `pop()` should block if there is no message available. Let us first see the monitor implementation:

```
public class DiscardBuffer {
        private Object buffer;
        synchronized void push (Object o) {
                buffer = o;
                notify();
        }
        synchronized Object pop () throws Exception {
                while (buffer!=null)
                        wait();
                Object r = buffer;
                buffer = null;
                return r;
        }
}
```

Here, `buffer==null` is true only when there is no report to process. Repeated calls to `push(o)` will overwrite messages with the latest data. Calls to `pop()` wait until a report is available, then take the report, write the buffer to `null` (which stops other computers Processing the same data), and returns the report to whichever ship's computer called `pop()` for processing.

Now we compare this to an implementation of the same buffer, but purely using chords. Because we have to implement precisely the semantics of `wait()` and `notify()`, the translation is much like the result of φ_{JS+}, only with some optimisations. The translation is shown in figure 9.2.

We can see that this, much like the implementation of the active object with monitors, is much larger and thus harder to understand and maintain.

We conclude that different applications would benefit from implementation in either chords or monitors, depending on the behaviour required. If the behaviour is closer to the semantics of chords, we should implement it in chords. Likewise there are times when we should use monitors. It seems the implementation of an ordered "fifo" buffer is of similar complexity in both paradigms:

```
public class UnboundedOrderedBuffer {

        private List list = new LinkedList();

        synchronized void push (Object o) {
                list.add(o);
                notify();
        }

        synchronized Object pop () throws Exception {
                while (list.size()==0)
                        wait();
                return list.remove(0);
        }
}
```

```
public class UnboundedOrderedBuffer {

        private block() & pulse() { }

        private List list = new SynchronizedLinkedList();

        void push (Object o) {
                list.add(o);
                this.pulse();
        }

        Object pop () {
                this.block();
                return list.remove(0);
        }
}
```

```
public class DiscardBuffer {

        private lock() & unlock() { }

        private block() & pulse() { }

        private int counter;

        private Object buffer;

        void push (Object o) {
                this.lock();
                buffer = o;
                if (counter!=0) {
                        this.pulse();
                        this.counter--;
                }
                this.unlock();

        }

        Object pop () throws Exception {
                this.lock();
                while (buffer!=null) {
                        this.counter++;
                        this.unlock();
                        this.block();
                        this.lock();
                }
                Object r = buffer;
                buffer = null;
                unlock();
                return r;
        }

}
```

Figure 9.2: Chorded translation of the discard buffer (ship's radar) example

9.3 Do chords discourage mistakes? An analogy with garbage collection.

The following remark is from [4]:

> "Chords might provide the same kind of productivity gains for programmers as automatic garbage collection has in the past."

Using the expressiveness results from this report, we can investigate this. Let us review the productivity gains that we can attribute to garbage collection in programming languages.

A language with garbage collection does not support the "freeing" of allocated memory. To convert a program that manually freed its allocated memory, we just need to remove all the calls to `free(memory)`, perhaps replacing them with `voidval`. The behaviour remains the same because the memory is freed by the run-time environment. Assuming the program that did the manual freeing was correct, it did not keep references to the freed memory, and did not fail to free memory to which all references had been lost. In this case, in the translated program the garbage collector is doing the same job as the program did, so the behaviour is the same.

Suppose the programmer made an error in their programming, and this was not spotted. Suppose an area of memory continued to be used after it was freed. On all the test runs, the behaviour was correct, but there was always the possibility of something going wrong. Formally, execution of the program would have resulted in some kind of run-time exception, but concrete implementations often do not check for errors, because this undermines performance.

If we convert this program into a language with garbage collection, by simply removing all the `free()` calls, we actually end up with a program with different behaviour, and no errors. Re-writing a program with garbage collection can remove errors from the program.

This is not the case with our translations here. When we convert monitor programs so that they use chords, the behaviour is always the same, including race conditions and deadlock. Whereas garbage collection completely eliminates the possibility of bad behaviour, chords merely allow us to express this bad behaviour in a different way. For example, chorded programs can deadlock if two threads are both waiting for the other to send a message.

If, however, we had a static analysis that detected deadlock in programs, and these programs were declared mal-formed, this would likely provide the same kind of productivity gains for programmers as garbage collection has done. We cannot "fix" deadlock, like we can with bad memory management, but we can at least warn about it.

Chords may however discourage mistakes in concurrent programming, because as seen above, some programs can be written much more concisely with chords. Clear and concise programs are easier for the programmer to intuitively verify, and this should in itself discourage mistakes. The question is: How many programs will need to re-implement chords, or chord-like message passing mechanisms in their code. If there is enough demand for chord-like synchronisation primitives, then their inclusion in a language will have a big impact on the correctness of code at large.

In summary garbage collection makes memory errors impossible. Chords have a complex semantics and this might draw complexity out of programs, making them simpler.

Chords therefore might have a similar impact to object-oriented programming, since in languages like C, one has to implement inheritance and late binding with structures and function pointers, and this can invite errors. Whether the impact on the programming community will be as massive as object oriented programming was, will depend on how applicable the chord semantics are within the programs that need to be written.

9.4 Where did chords make the encodings difficult?

In the process of inventing translations that encode synchronisation with chords, we have put a lot of 'stress' on chords, as a mechanism for expressing synchronisation in an imperative object-oriented environment. The encodings show the limitations of chords, i.e. how they can occasionally be slightly clumsy when expressing synchronisation.

In all the translations to chorded languages (φ_{AS}, φSA, φ_{JS+}), new methods had to be created to represent the required synchronisation behaviour. This is not always appropriate because the methods have to be added at the root of the class definition, and this may be some distance from where the synchronisation needs to take place.

For instance, in φ_{SA} we had to create a large number of locks per class, one for each `spawn` site. These locks would build up in the root of the class and this would adversely affect the clarity of the code. Forcing the programmer to do synchronisation via method calls is a weakness of chords. The programmer should have more options for expressing synchronisation.

What does synchronisation have to do with method invocation anyway? In the Join calculus, there was no imperative state, so code was written in a functional style, with use of recursion. It was therefore natural for synchronisation to be coupled with method invocation because there was nowhere else to put it.

In a language like Java or $C^\sharp$ where the code is much more complex, and a lot of different behaviours are being expressed using different constructs, it is more important to keep code tidy. Monitors manage this nicely since the synchronisation happens in the method bodies. Chords, on the other hand, can clutter the top-level of the class with irrelevant information.

Another problem with concurrency with chords is that the queues often need initialising. For example, the chords idiom that implements a mutex has to have an initial asynchronous message sent to `unlock()`. This might be a problem because the constructor is some distance from where the synchronisation might typically occur. The separation of related concepts in program code makes maintenance hard, since changes to one part of the code can happen while neglecting the other part.

When we defined structure preservation in 3.1.2, we allowed translations to add methods and constructors, because these changes are not as serious as changes to the syntactic structure of source expressions. We still feel that chords can be slightly clumsy when expressing synchronisation, maybe the Join calculus needs more adaptation to fit neatly in an imperative object-oriented language like Java or $C^\sharp$.

Chapter 10

Evaluation and further work

10.1 Evaluation

We now criticise the approach used in this report. There are two kinds of criticisms – "Are the formalisms and proofs correct?" and "Are the models realistic?".

10.1.1 Criticisms of formal work

While we have taken care to formally prove certain properties, the proofs have not been checked by a theorem prover, so we cannot be absolutely sure that there are no missing cases or invalid assumptions. We cannot be sure that the encodings do indeed preserve behaviour, because we have not formally proved this. This was omitted due to time constraints, and the fact that it is non-trivial to even formally state the property of behaviour-preservation. More research into bisimulation and observational equivalence was required and there was no time.

The type systems used were not sufficient to ensure programs did not "go wrong". We also did not prove the type system has the subject reduction property, since this would have required the introduction of run-time well-formedness and the typing of the run-time expressions.

It would have been good to have proven progress of well-formed programs, indeed this might have been a necessary assumption for a proof of preservation of behaviour over some of the translations. For example when encoding Java with chords φ_{JS+} (chapter 7), it is assumed that we always have the lock when we use the functions `wait` **e**, `notify` **e**, and `notifyAll` **e**, and $\mathcal{L}^J$ programs behave differently to their $\mathcal{L}^{S+}$ counterparts when this assumption is broken.

10.1.2 Accuracy of formalisms

The formalised languages are necessarily abstract, this makes the proofs and translations easier, but means the results are more 'distant' from real programming languages. We not only omitted features like exceptions and arithmetic which are orthogonal to synchronisation, but also relevant features like the ability to interrupt the blocking of a thread, and the ability to test the state of a mutex without blocking, which is often implemented as a `try_lock()` function.

Perhaps the least convincing aspect of this work was the extension of the property of structure-preservation from [5], to include the structure of programs. Allowing the addition of methods to classes during a translation seems like an arbitrary decision, and the justification for doing so was very subjective.

In fact, we believe even the original definition was slightly flawed: Consider the encoding of `notifyAll()` using `wait()` and `notify()`. This is possible if we wrap the calls to `wait()` and `notify()` with functions that maintain a count of how many threads are currently waiting. Then the call to `notifyAll()` can just call `notify()` the correct number of times to wake up all the threads. The question is, is this structure preserving. Certainly, all the calls to `wait()` need to be replaced with calls to the wrapper function, or they wont be accounted for by the counter.

If we consider the comparative expressive power of two languages, one with `wait()`, `notify()`, and `notifyAll()`, and the other with just `wait()` and `notify()`, then `wait()` and `notify()` are shared program constructs. This means the translation given above is not structure-preserving since it is not homomorphic over `wait()` and `notify()`.

If, however, we consider the comparative expressive power of two languages, one with `wait()`, `notify()`, and `notifyAll()`, and the other with `wait2()` and `notify2()`, i.e. two constructs with the same semantics as above, but just differently named, then there are no shared constructs, and the translation *is* structure-preserving. It does not seem clear whether it is wrong to consider them as differently named constructs, it all depends on how the language is formalised.

If we compare two very similar languages, $\mathcal{L}^1$ and $\mathcal{L}^2$, the structure-preservation property is very strict - it must be homomorphic over the majority of language constructs. But if we go via another language $\mathcal{L}^x$ and back again, a language that has no constructs in common with either $\mathcal{L}^1$ or $\mathcal{L}^2$, then we have much more liberty with our translations $\mathcal{L}^1 \leftrightarrow \mathcal{L}^x$ and $\mathcal{L}^x \leftrightarrow \mathcal{L}^2$. Thus $\mathcal{L}^1$ may be equivalent to $\mathcal{L}^x$, and $\mathcal{L}^x$ may be equivalent to $\mathcal{L}^2$, but $\mathcal{L}^1$ may not be equivalent to $\mathcal{L}^2$, in terms of expressiveness. In other words, the expressiveness relation may not be transitive!

Also, some programs which are considered to have the same structure are actually quite different. This calls into question whether the formal definition of the structure-preservation property is a realistic enough model of what a programmer would think of as an acceptable change to a program's structure, when porting that program from one language to another.

For example, when translating a construct that is not shared between the two languages, the result must be a macro encoding of the sub-expressions in that construct. The macro itself can be arbitrarily large, however. It just has to be consistent across all instances of that construct in the source program, i.e. the code that replaces the construct cannot depend on that constructs sub-expressions, except to embed them within itself. If the macro was very large, it could conceivably dwarf the rest of the program. Having said this, the translations in this report tended to be quite small.

10.2 Further work

Of the above issues, several could be addressed by further work. It would be useful to prove that the translations preserve behaviour, and also to develop the semantics and

well-formedness judgement so that it excludes programs that do not progress. It would be interesting to investigate the expressive power of the synchronisation constructs that we omitted to include in our formalisms, such as timed waits, interrupted waits, and testing the state of the mutex without blocking.

It might be better to consider structure-preservation as a graded measure, rather than an absolute judgement. We can consider precisely how much structure can be preserved in translations, and even see what changes to the semantics of chords would be needed to maximise the preserved structure, in the translations. For instance, we can subtract "points" if a translation needs to add classes, add methods to a class, or add code to the constructor of a class.

Chords are a very specific kind of message passing, the queues are unordered, and unbounded. Other approaches are also possible. Having unbounded queues could be a problem when one thread is sending messages faster than other threads are consuming them. It would be interesting to compare the different approaches, and even repeat this work but with an extended version of chords that allow the programmer to specify these details.

For instance, the programmer could specify whether the queues were to be ordered or unordered, whether the queue is bounded, if it *is* bounded, whether to block, or discard messages that are sent to a full queue. One could also add features to flush a queue, or to inspect the emptiness of a queue like `try_lock()` inspects the lockedness of a mutex without blocking.

We could introduce a mechanism where threads can be interrupted while they are blocked at a synchronous method call. We could let the programmer initialise a queue outside of the constructor, like the way that fields can be initialised to a default value outside of the constructor. These extensions would make chords more flexible, and may even make monitors completely redundant, i.e. a particular configuration of a chord might look and behave exactly like condition variables. This is like the way that if a method has chords, it does not need methods because a synchronous chord with no asynchronous parts is identical to a method in every way. There are also other kinds of message passing in other languages, like in Concurrent ML, and in Haskell. One could investigate whether chords can behave like these language constructs.

It would be interesting to decouple chords from methods, i.e. have expression syntax to send and receive messages. Sending a message would be like an asynchronous method invocation, and receiving a message would be like invoking a synchronous chord, i.e. it would block until all the queues were non-empty, then proceed. Not only would this allow the programmer to keep the synchronisation code out of the root of the class definition, it would also avoid the inheritance anomaly. We can call these chord-like constructs "microchords". It would be easy to implement conventional chords on top of them, just by receiving a message at the top of a function that we want to behave like a synchronous chord.

There is also a link between condition variables, and mutual exclusion. This is evident in the definition of φ_{JS+}, where the same chord mechanism implements both the mutex, and the internals of the condition variable functionality. This could benefit from further study because we might be able to make the monitor formalisation even simpler, by showing that one can be implemented one on top of the other. It might be possible to show there is

an even lower-level and simper formalisation of synchronisation, on top of which monitors can be implemented.

There is much more to consider, when deciding what synchronisation primitives to offer the programmer. Ultimately, we must present enough power to make programs as simple and clear as possible, without introducing too much complexity. The expressiveness of synchronisation constructs, as considered in this report, tells us something of their power, and from this we are able to better understand the effect of language on the programming process. However, subtle issues such as the conciseness of often-written programs when expressed in different languages are also important.

References

[1] Alexander Ahern and Nobuko Yoshida. Formalising java rmi with explicit code mobility. To appear at OOPSLA 2005, 2005.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for $c^\sharp$, 2002.

[3] developerWorks. More flexible, scalable locking in jdk 5.0, October 2004. `http://www-128.ibm.com/developerworks/java/library/j-jtp10264/`.

[4] Sophia Drossopoulou, Alexis Petrounias, Alex Buckley, and Susan Eisenbach. School: a small chorded object-oriented language. Submitted for publication, March 2005.

[5] Matthias Felleisen. On the expressive power of programming languages. In *ESOP '90: Selected papers from the symposium on 3rd European symposium on programming*, pages 35–75, Amsterdam, The Netherlands, The Netherlands, 1991. Elsevier North-Holland, Inc.

[6] Cedric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM Press.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[8] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[9] International Organization for Standardization. *Information technology–Portable Operating System Interface (POSIX)*. 1990. International standard ISO/IEC 9945. IEEE Std 1003.1-1990 (revision of IEEE Std 1003.1-1988). Part 1. System application program interface (API) [C language].

[10] Xavier Leroy. `pthread_mutexattr_init(3)`. Manual page for LinuxThreads - an implementation of posix threading.

[11] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. In *Dr. Dobb's Journal*, March 2005. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[12] Tim Wood. A chorded compiler for java, June 2004. MEng thesis. Supervisor – Susan Eisenbach.